VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Masivně paralelní implementace nedeterministických konečných automatů

Massive Parallel Implementation of Nondeterministic Finite Automata

2011 Michal Holiš

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6. května 2011 .....

Rád bych na tomto místě poděkoval všem, kteří mi pomohli, protože bez nich by tato práce nevznikla.

Abstrakt

V této diplomové práci se zabýváme implementacemi různých algoritmů na grafických kartách s využitím technologie CUDA. Prozkoumali jsme oblasti zpracování řídkých matic a provádění konečných nedeterministických automatů. Z oblasti zpracovávání řídkých matic jsme se zaměřili na násobení řídké matice vektorem, násobení dvou řídkých matic a na výpočet matice podobnosti. V průběhu práce jsme zjistili, že ne všechny algoritmy jsou pro technologii CUDA vhodné. Na závěr jsme naimplementované algoritmy experimentálně ověřili.

Klíčová slova: nVidia CUDA, Zpracování řídkých matic, Cannonův algoritmus, Matice podobnosti, Nedeterministické konečné automaty, Vyhledávání v řetězci s chybou

Abstract

This thesis focuses on implementations of various algorithms using graphics cards with CUDA technology. We have explored processing of sparse matrices and execution of nondeterministic finite automata. From area of processing sparse matrices we have focused on multiplication of sparse matrix with dense vector, multiplication of two sparse matrices and computation of similarity matrix. We have found, that not all of the algorithms are suitable for CUDA technology. Finally we have experimentally verified all implemented algorithms.

Keywords: nVidia CUDA, Processing sparse matrices, Cannon's algorithm, Similarity matrix, Nondeterministic finite automata, Approximate string matching with differences

Seznam použitých zkratek a symbolů

ALU	–	Arithmetic logic unit
CCS	–	Compressed Column Storage
CRS	–	Compressed Row Storage
MIMD	–	Multiple Instruction Multiple Data
MMX	–	Multi Media eXtensions
MPI	–	Message Passing Interface
NFA	–	Nondeterministic Finita Automata
PVM	–	Parallel Virtual Machine
SIMD	–	Single Instruction Multiple Data

Obsah

Seznam obrázků

Seznam tabulek

Seznam algoritmů

1 Úvod

V diplomové práci se zabýváme konkrétními implementacemi algoritmů, řešících několik problémů za pomocí technologie CUDA (více informací o CUDA např. v [1]).

CUDA je paralelní výpočetní architektura vyvinutá firmou nVidia. Naším cílem je tuto platformu prozkoumat a pokusit se jí využít k urychlení výpočtů ve dvou hlavních oblastech.

První oblastí je zpracování řídkých matic, konkrétně nás zajímá násobení řídké matice hustým vektorem a násobení dvou řídkých matic. Na prvním problému prozkoumáme jednotlivé formáty uchovávání řídkých matic a postavíme základ pro násobení dvou matic. Vyřešení tohoto problému je velmi zajímavé zejména z toho důvodu, že v době psaní této práce pro něj neexistuje implementace ani v nejznámější knihovně pro zpracování řídkých dat CUBLAS. Rovněž jsme se zaměřili na výpočet matice podobnosti pro řídkou matici.

Druhou oblastí, kterou zde prozkoumáme, je implementace obecného konečného nedeterministického automatu na této technologii a sestavení konkrétního automatu pro vyhledávání v řetězci s chybou.

Platforma CUDA je poměrně novou paralelní architekturou a díky její specifičnosti je řešení některých problémů s její pomocí velmi obtížné. Na mnoho těchto problémů jsme narazili a snažili jsme se je efektivně řešit, všechny budou rozebrány v jednotlivých kapitolách.

V průběhu tvorby této práce vzniklo několik aplikací, které jsou napsány v CUDA v jazyce C a dvě podpůrné aplikace na platformě .NET Framework v jazyce C#.

S hlavními, zde prezentovanými algoritmy, jsme pochopitelně provedli patřičné experimenty a jejich výsledky zde uvádíme.

Struktura práce

Práce se dělí na devět základních kapitol. V kapitole 1 jsme stručně popsali úvod a základní motivaci pro tuto práci. V 2. kapitole se věnujeme základnímu popisu samotné architektury nVidia CUDA a ukážeme její silné i slabé stránky. Ve 3. kapitole se věnujeme násobení řídkých matic, probereme možnosti implementace násobení řídké matice hustým vektorem i násobení dvou řídkých matic. Rovněž se zaměříme na výpočet matice podobnosti. Kapitola 4 se zabývá experimenty a zhodnocením naměřených výsledků algoritmů prezentovaných ve třetí kapitole. Kapitola 5 pokrývá samotné stěžejní téma, které jsme chtěli touto prací prozkoumat, a tím je implementace vykonávání obecného nedeterministického konečného automatu. V 6. kapitole ověříme experimenty naši implementaci konečného automatu. Kapitolu 7 věnujeme popisu přiložených testovacích a podpůrných programů. V kapitole 8 pak shrnujeme výsledky této práce a uvádíme další možnosti pro pokračování ve zde načatých tématech. Poslední kapitolu 9 věnujeme referencím.

2 Úvod do nVidia CUDA

CUDA je rozhraní vytvořené firmou nVidia určené k masivně paralelním výpočtům realizovaným na moderních grafických kartách této firmy. Avšak nazvat CUDA pouhým programovým rozhraním by bylo poněkud nepřesné a neúplné. CUDA velmi úzce souvisí se změnou, která nastala v architektuře samotných grafických akcelerátorů.

Dříve jsme byli zvyklí na tradiční rozdělení akcelerátorů na dvě hlavní výpočetní oblasti - na vertex shadery a na pixel shadery. Popis této architektury můžeme najít např. na webové stránce MSDN [2]. Ačkoli i na této platformě bylo možné určité výpočty provádět, prostředí bylo velmi omezené a na veškeré řešené problémy bylo potřeba nahlížet jako na grafické problémy a zároveň znát některý z jazyků pro shadery určený jako je např. OpenGL GLSL nebo Microsoft HLSL. Více o OpenGL GLSL se lze dočíst na domovské stránce [3], o Microsoft HLSL pak např. na MSDN [4].

CUDA přišla s alternativní architekturou, která je založena na unifikovaných shaderech, která umožňuje využívat kteroukouliv ALU na grafické kartě k obecným výpočtům. Tyto jednotky byly hardwarově navrženy tak, aby dodržely standard IEEE 754-2008 (32-bitová čísla s plovoucí desetinnou čárkou) a umožnily vykonávání instrukcí zaměřených na zpracování obecných výpočtů namísto operací specifických pouze pro grafické výpočty. Je možné provádět i výpočty s čísly s dvojtou přesností (64-bit) ale jejich chování se od standardu IEEE 754 odlišuje. CUDA je SIMD architekturou (narozdíl např. od připravované MIMD architektury Larrabee firmy Intel), více informací o CUDA lze nalézt např. na webové stránce [1].

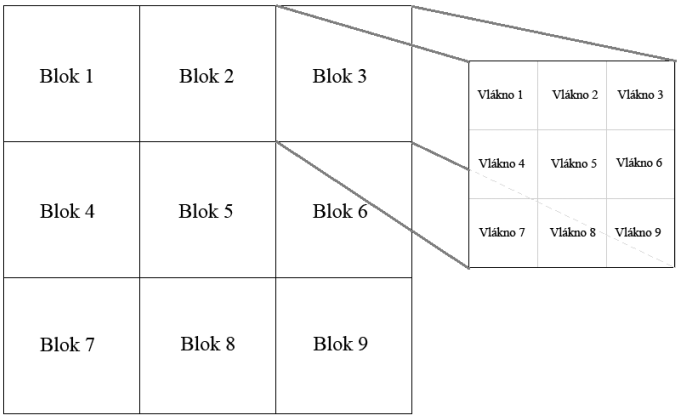
CUDA zároveň přišla s vlastním programovacím jazykem založeným na jazyku C, který obohatila o několik klíčových slov určených k využití a označení speciálních funkcí souvisejících s grafickým akcelerátorem.

2.1 Vývoj v nVidia CUDA

Nejdůležitějším základem pro vývoj v nVidia CUDA je tzv. *kernel*. Na kernel můžeme nahlížet jako na speciální funkci v jazyce C, která bude vykonávana na grafické kartě. Pomocí tohoto kernelu budeme rozdělovat výpočetní práci mezi jednotlivé ALU grafické karty.

Při paralelizaci většiny problému budeme postupovat pomocí datové blokové dekompozice, která je pro SIMD architekturu nejvhodnější. Rozdělíme tedy úlohu na jednotlivé bloky dat, které mohou být zpracovávany na sobě nezávisle a pro každý takovýto blok bude poté na grafické kartě jednou zavolána kernel funkce a provede nad těmito daty vždy stejný výpočet.

Tento kernel bude spuštěn z hlavního procesu našeho programu běžícího na CPU (označován jako *hostující* proces) a budou mu předem poskytnuta data, nahrána do paměti grafické karty. Po skončení výpočtu budeme mít možnost načíst data zpět z grafické karty do paměti hostujícího procesu. Spuštění samotného kernelu probíhá sice asynchronně, ale vykopírování výsledků z paměti grafické karty je synchronní operací. Tento fakt je velmi důležitý pro návrh paralelních programů na platformě CUDA. Zejména



Obrázek 1: Architektura nVidia CUDA - bloky a vlákna

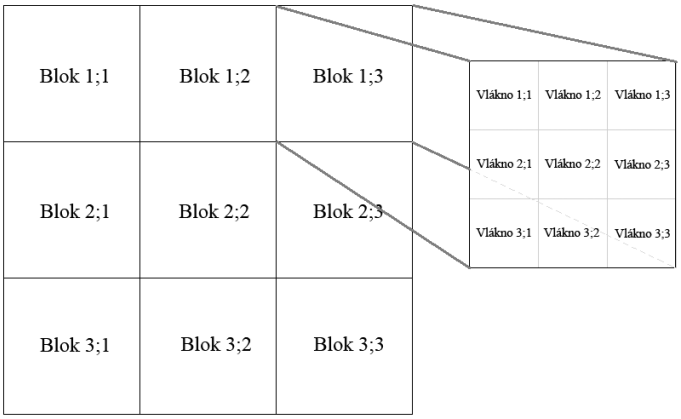
nemožnost v průběhu výpočtu dynamicky průběžně získávat data z grafického akcelerátoru je jedním z větších faktorů limitujících naše možnosti při návrhu algoritmu.

Víme už, že při spouštění výpočtu budeme data rozdělovat do bloků a specifikovat počet procesů, které se na grafické kartě spustí. Počet procesů se na nVidia CUDA určuje specifickým způsobem.

V základu můžeme říci, že chceme spustit n procesů. Každý takto spuštěný proces se v CUDA nazývá blok. Tento blok se dále dělí do m vláken. Rozdělení můžeme vidět na obrázku 1. Počet vláken v rámci jednoho bloku je velmi výrazně omezen, v současné době se jedná o maximální počet 512 vláken na jeden blok. Pokud tedy budeme chtít spustit n procesů máme několik možností. Můžeme spustit n bloků, každý s jedním vláknem. Nebo můžeme spustit $\frac{n}{10}$ bloků, kde v rámci jednoho bloku poběží 10 vláken. V čem je rozdíl? Ten spočívá hlavně v možnosti komunikace. Mezi jednotlivými bloky totiž žádná komunikace neexistuje. Žádné vlákno nemůže jakýmkoliv způsobem ovlivnit vlákno z jiného bloku. Oproti tomu vlákna v jednom bloku mohou mezi sebou komunikovat pomocí takzvané sdílené paměti. Více o volání kernelu si můžeme přečíst v "CUDA Programming Guide" [5].

Na obrázku 1 vidíme bloky a vlákna označená jednoduchým číslem. V rámci běhu kernelu vždy obdržíme toto ID a podle něj si určíme oblast dat, nad kterými budeme provádět naše výpočty. CUDA umožňuje pro naši lepší orientaci označovat vlákna namísto jednoduchého čísla až trojrozměrným vektorem. Toto nemá vliv na způsob jakým se budou vlákna spouštět, ale je to přínosné v případě, že chceme pracovat např. s dvojrozměrnými maticemi, určitě pro nás bude snažší pracovat s daty, když nám v každém běhu kernelu bude sdělena dvojrozměrná poloha v datech než jednoduchý index, který bychom do dvojrozměrné podoby museli pokaždé přepočítávat. Příklad dvojrozměrné indexace můžeme vidět na obrázku 2.

Důležitým pojmem, který musíme uvést je tzv. *warp size.* Při spuštění kernelu na multiprocesoru grafické karty jsou běžící vlákna zorganizovány do bloků, v rámci kterých



Obrázek 2: Architektura nVidia CUDA - bloky a vlákna - dvojrozměrné uspořádání

vykonávají všechny vlákna stejnou instrukci. Tyto bloky se nazývají *warpy*. Maximální počet vláken v jednom takovémto *warpu* se označuje jako *warp size*.

2.2 Implementační omezení

CUDA jako platforma disponuje velmi silným výkonostním potenciálem, ale naráží hned v několika případech na svou nízkou flexibilitu a malé množství podporovaných základních operací.

Už sama architektura SIMD je velmi omezejucím prvkem. SIMD samotné není překvapením, historicky se tato architektura vždy objevovala ve spojení s grafickými operacemi v reálném čase, ať už mluvíme o straně grafických akcelerátorů nebo např. o rozšíření MMX. Ze samotného principu fungování grafických algoritmů, které se pro operace v reálném čase omezují povětšinou na jednotné operace nad texturami reprezentujících různá data, je tato architektura správnou volbou. Pro naše výpočty bychom ale určitě uvítali architekturu MIMD, která by nám dala mnohem větší prostor pro návrh algoritmů.

Dále je problémem synchronizace a komunikace mezi jednotlivými bloky/vlákny. Výše byla zmíněna komunikace za pomoci sdílené paměti, která je však možná pouze mezi vlákny sdílejícími jeden blok. Také zde narážíme na synchronizaci mezi jednotlivými vlákny, která prakticky neexistuje. Jediný nativně podporovaný způsob je jednoduchý souběh vláken před čtením a zápisem do sdílené paměti.

Naštěstí bylo podpořeno několik jednoduchých atomických operací, které jsme při implementacích využili. Zejména je podpořena operace atomického součtu.

Pokročilejší synchronizaci je u CUDA často diskutované téma. Nejčastěji se zmiňují implementace kritické sekce s využitím atomických operací na principu *spinlocku*. Toto řešení však v praxi selhává v případě, že chceme spustit větší počet bloků než je počet multiprocesorů a může samo o sobě vést k deadlocku. O principu fungování *spinlocku* se lze více dočíst na webové stránce projektu “Concurrency Kit” [6], který se zabývá implementacemi různých podpůrných rutin pro usnadnění vývoje paralelních programů.

Také tím, že je CUDA poměrně mladou platformou, obsahuje občas chyby bránící v jejím plném využití. Při tvorbě této práce jsme narazili např. na neschopnost alokovat na některých kartách větší bloky paměti, nemožnost kopírovat na grafickou kartu data v cyklu, vysokou chybovost samotného debuggeru apod.

Je nutno ale říci, že na vývoji CUDA se neustále pracuje a většina chyb uvedená v předchozím odstavci byla ještě před dokončením této práce opravena. Je možné, že se s narůstající popularitou této platformy rozroste funkcionalita základu a nabídne mnohem flexibilnější vývoj.

I přes některé chyby je platforma CUDA dostatečně zajímavá z důvodu velkého výkonostního potenciálu, který, vzhledem k velkému počtu výpočetních jader, nabízí.

2.3 Příklad implementace

Pro doplnění představy o vývoji v technologii nVidia CUDA uvádíme v této kapitole krátkou ukázku jednoduchého programu pro sečtení dvou polí.

Na sestavě, na které budeme vyvíjet, potřebujeme grafickou kartu podporující nVidia CUDA, seznam těchto karet je k nalezení na [7]. Pro vývoj v operačním systému Windows budeme dále potřebovat nVidia CUDA Toolkit, tento můžeme stáhnout na webu výrobce [8]. Pro zkompilování vytvořeného programu jsme použili Microsoft Visual Studio 2008. V nových verzích nVidia CUDA Toolkit je k dispozici, při výchozím nastavení instalace, v adresáři *[Disk]:\ProgramData\NVIDIA Corporation\NVIDIA GPU Computing SDK [Verze]\C\src\template* vzorová solution, která obsahuje dva základní soubory: *template.cu* a *template_ kernel.cu*. V prvním jmenovaném je obsažen kód pro hostující proces a do druhého souboru se zapisuje kód samotného kernelu. Tato šablona nahrazuje standardní kompilátor Visual Studia kompilátorem pro CUDA, který přeloží části specifické pro CUDA do konstrukcí jazyka C, výsledek je potom celý přeložen standardním kompilátorem jazyka C.

Na algoritmu 1 vidíme okomentovaný výpis kódu našeho kernelu pro součet dvou vektorů. Tento kód bude spuštěn jednou pro každý prvek těchto vektorů a provede jeden součet. Toto bude jediná funkce, kterou budeme potřebovat uvést v souboru *template_kernel.cu*.

Kernel je velmi jednoduchý, v prvním kroku pouze spočítá index do pole dat, který určí dvojici čísel, které bude sčítat. V druhém kroku provede samotné sečtení. Vidíme, že se kromě klíčového slova *global* jedná o standardní funkci.

Algoritmus 1 CUDA příklad - kernel

```
__global__ void add(
    float *x,
    float *y,
    float *output,
    int num_items)
{
    // V proměnné threadIdx je vždy uložen vektor identifikující id spuštěného vlákna. Pro více
        než 512 spuštěných vláken musíme počítat s ID bloku, které je uloženo v proměnné
        blockIdx. Počet vláken v jednom bloku je pak uložen v proměnné gridDim.
    int index = blockIdx.x * gridDim.x + threadIdx.x;
    // Vzhledem k systému, jakým se určuje index do dat, může dojít k tomu, že budeme mít index
        ukazující mimo pole, v případě, že se toto stane, ukončíme aktuální vlákno.
    if (index >= num_items) return;
    // Provedeme sečtení dvou prvků z vektorů x a y a přiřadíme výsledek do výstupního pole output
    output[index] = x[index] + y[index];
}
```

Na algoritmu 2 pak uvádíme kód pro hostující proces. Nejprve potřebujeme na grafickou kartu nahrát data, nad kterými bude operace probíhat. Data si standardním způsobem vytvoříme na hostujícím procesu, v našem případě jsme vytvořili dva náhodné vektory a jeden nulový vektor pro výsledek.

Poté si vytvoříme ukazatele na tyto data na grafické kartě. My jsme si tyto proměnné označili prefixem *d_*, proměnná *d_x* bude tedy ukazovat na data vektoru *x* v paměti grafické karty. Tento ukazatel je v rámci hostujícího procesu neplatný a nemůžeme s ním tedy žádným způsobem pracovat. Můžeme jej používat pouze jako parametr do funkcí pracujících s pamětí grafické karty.

Pro alokování paměti na grafické kartě je určena funkce *cudaMalloc*. Tato funkce funguje obdobně jako standardní funkce *malloc*.

Parametry *cudaMalloc*:

- *devPtr* - prvním parametrem musí být ukazatel do paměti grafické karty,
- *size* - druhým parametrem je počet bajtů, které budou alokovány.

Pro kopírování dat je připravena funkce *cudaMemCpy*. Tato funkce funguje opět obdobně jako klasická funkce *memcpy*.

Parametry cudaMemCpy:

- *dst* - jako první parametr obdrží ukazatel na místo v paměti na které budou zkopírována data,
- *src* - ukazatele určující zdrojová data, která budou zkopírována,
- *count* - počet bajtů, které budou zkopírovány,
- *kind* - enumerace, která určuje směr kopírování, může nabývat jedné z následujících hodnot:

- *cudaMemcpyHostToHost* - z hostujícího procesu na hostující proces,
- *cudaMemcpyHostToDevice* - z hostujícího procesu na grafickou kartu,
- *cudaMemcpyDeviceToHost* - z grafické karty na hostující proces,
- *cudaMemcpyDeviceToDevice* - z paměti grafické karty do paměti grafické karty.

Až máme v paměti připravená veškerá data, můžeme spustit samotný kernel. Toto se provádí poměrně nestandardní syntaxí. Volání vypadá podobně jako volání funkce jazyka C, před seznam argumentů je ale přidána konstrukce <<<*grid, block*>>> která určí, kolikrát se daný kernel spustí. Proměnná *grid* zde reprezentuje 3D vektor určující počet spuštěných vláken v rámci jednoho bloku, proměnná *block* (opět 3D vektor) poté určuje počet těchto bloků, které budou spuštěny.

Po skončení běhu kernelu můžeme výsledek zkopírovat do paměti hostujícího procesu a program ukončit.

Algoritmus 2 CUDA příklad - hostující proces

```
// Dvě následující funkce vypočtou velikost bloku a gridu pro počet požadovaných vláken. V
    případě, že chceme méně než 512 vláken, stačí nám blok o velikosti (1,1,1) a grid o velikosti
    (počet_vlaken, 1, 1). Jinak je vygenerován grid o velikosti (512, 1, 1) a blok s patřičnou
    velikostí tak, aby jeho vynásobením s velikostí gridu vzniklo číslo větší než je požadovaný
    počet vláken. V tomto příkladu používáme pouze indexaci za pomocí x−ové složky vektoru ID,
    v jiných případech by jsme mohli využít i vícerozměrných indexací.
dim3 make_block(int num_items)
{
    if (num_items <= 512)
        return dim3(num_items, 1, 1);
    return dim3(512, 1, 1);
}

dim3 make_grid(int num_items)
{
    if (num_items <= 512)
        return dim3(1,1,1);
    double dimension = num_items / 512;
    dimension++;
    return dim3((int)dimension, 1, 1);
}

int main(int argc, char** argv)
{
    // Definice dat na hostujícím procesu.
    const int size = 5;
    float *x = (float*)malloc(sizeof(float) * size);
    float *y = (float*)malloc(sizeof(float) * size);
    float *output = (float*)malloc(sizeof(float) * size);
    // Provedeme naplnění proměnných x, y daty. Znulujeme vektor output
    ...
    // Definujeme ukazatel na vektor X uložený v paměti grafické karty.
    float *d_x;
    // Naalokujeme potřebný prostor v paměti grafické karty.
    cutilSafeCall (cudaMalloc((void**) &d_x, sizeof(float) * size));
    // Nakopírujeme vektor na grafickou kartu.
    cutilSafeCall (cudaMemcpy(d_x, x, sizeof(float) * size, cudaMemcpyHostToDevice));

    // Stejným způsobem nakopírujeme i vektor Y a vektor pro výsledek.
    float *d_y;
    cutilSafeCall (cudaMalloc((void**) &d_y, sizeof(float) * size));
    cutilSafeCall (cudaMemcpy(d_y, y, sizeof(float) * size, cudaMemcpyHostToDevice));

    float *d_output;
    cutilSafeCall (cudaMalloc((void**) &d_output, sizeof(float) * size));
    cutilSafeCall (cudaMemcpy(d_output, output, sizeof(float) * size, cudaMemcpyHostToDevice));

    // Vyvoříme block a grid.
    dim3 grid = make_grid(size);
    dim3 block = make_block(size);

    // Spustíme kernel
    add<<<grid, block>>>(d_x, d_y, d_output, size);

    // Vykopírujeme výsledek z grafické karty
    cutilSafeCall (cudaMemcpy(output, d_output, sizeof(float) * size, cudaMemcpyDeviceToHost));
    // Ukončíme běh programu
    cudaThreadExit();
}
```

3 Násobení řídkých matic

Řídká matice je speciální třída matic, která se vyznačuje tím, že část prvků, které obsahuje se rovná nule. Řídké matice jsou dnes výsledkem mnoha výpočetních operací a jejich následné zpracování je součástí mnoha algoritmů. Zároveň toto zpracování bývá ve většině případů částí výpočetně nejsložitější a její zefektivnění tedy často vede k znatelnému zlepšení časové složitosti daného problému. Některé algoritmy využívající řídké matice můžeme najít v knize [9] nebo v knize [10].

V první části této sekce předvedeme možnou techniku provedení násobení řídké matice hustým vektorem. Tato nám poslouží hlavně k výběru datové reprezentace řídké matice vhodné k použití na platformě CUDA.

V druhé části se budeme věnovat hlavnímu problému, který je stěžejním tématem této sekce, a tím je násobení dvou řídkých matic. S tímto problémem se potýkáme při řešení mnoha výpočetních problémů, jednou z motivací pro urychlení řešení tohoto problému pro nás byl např. výpočet matice podobnosti. Urychlením tohoto časově velmi náročného problému bychom docílili větší efektivity u některých shlukovacích algoritmů.

Zároveň se zvlášť podíváme i na alternativní implementaci násobení dvou matic, která bude sloužit pouze pro výpočet matice podobnosti pro velmi řídké matice.

3.1 Knihovna CUSPARSE

V průběhu sepisování této práce vznikla knihovna CUSPARSE, která je přímo podporována firmou nVidia a která obsahuje rutiny řešící problémy prezentované v kapitole 3.2. V době vytváření těchto algoritmů tato knihovna však ještě neexistovala.

I přes kvalitu a velký počet podporovaných operací v této knihovně, jí nejsou podporovány žádné algoritmy pro násobení řídké matice řídkou maticí. Algoritmy prezentované v sekcích 3.3 a 3.4 nejsou tedy ani v této největší knihovně pro zpracování řídkých dat na technologii CUDA přítomny.

Více o knihovně CUSPARSE lze vyhledat v dokumentaci [11] nebo v prezentaci na [12].

3.2 Násobení řídké matice hustým vektorem

Násobení řídké matice hustým vektorem je jedna z důležitých operací lineární algebry. Mnoho algoritmů, které tuto operaci využívají, včetně např. algoritmu pro nalezení vlastních čísel matice, je uvedeno v [9] nebo v knize [10]. V praxi se u tohoto násobení setkáváme s širokým spektrem rovnoměrně až nerovnoměrně uspořádaných matic s různou hustotou.

Prvním problémem při realizaci násobení je určení vhodné datové reprezentace řídké matice, která by byla co nejvíce paměťově efektivní a zároveň by nám poskytovala možnost násobení jednoduše zrealizovat.

Jak již bylo řečeno platforma CUDA je v některých ohledech velmi omezená a výběr vhodného formátu do velké míry určí konečnou efektivitu celého algoritmu. Tyto zkušenosti využijeme dále v práci, v kapitole 3.3.

Porovnání jednotlivých formátů bylo nastudováno z článku [13] srovnávajícího tyto datové formáty s ohledem na architekturu CUDA.

3.2.1 Formát COO

Formát COO je snad nejjednodušším možným formátem pro reprezentaci řídké matice. Matici reprezentuje za pomocí tří vektorů. Jeden vektor *data* obsahuje seznam všech nenulových prvků matice. Vektory *řádek* a *sloupec* pak obsahují indexy řádku a sloupce daného prvku matice. Tento formát je bohužel velmi paměťově náročný, jelikož pro n prvků je jeho paměťová složitost $3n$, vzhledem k tomu, že prvky mohou být náhodně seřazeny, bylo by velmi obtížné k násobení tento formát využívat, museli bychom vždy vyhledávat potřebný prvek k násobení v celé matici.

Formát COO tedy není k našemu účelu vhodný. I v případě, že by prvky seřazeny byly, stále je díky své paměťové náročnosti nepoužitelný. Pro určité aplikace, kde je důležité znát rychle přesně pozice všech prvků je však vhodný a i my jsme jej na několika místech k reprezentaci menších dat využili. V tabulce 1 vidíme vlevo hustou matici a vpravo její řídkou reprezentaci pomocí formátu COO.

7	*3*	0
0	*7*	0
8	0	*4*

řádek	0	0	1	2	2
sloupec	0	1	1	0	2
data	7	3	7	8	4

Tabulka 1: Struktura formátu COO

3.2.2 Formát DIA

DIA je zajímavý formát určený pro matice, které jsou tvořeny malým počtem nenulových diagonál. K uložení využívá jednoho pole nenulových prvků, které zároveň reprezentuje jednotlivé diagonály a jednoho pole odchylek, které určuje umístění diagonály. Záporné číslo značí diagonálu pod hlavní diagonálou, kladné číslo naopak určuje diagonálu nad hlavní diagonálou. V tabulce 2 je vlevo uvedena původní hustá matice. Uprostřed vidíme pole nenulových hodnot a vpravo pole odchylek, které určuje odsazení dané diagonály od diagonály hlavní. V případě, že matice je v tomto formátu reprezentovatelná, je násobení těchto matic vektorem jednoduchou záležitostí. Bohužel se zde snažíme o co nejobecnější reprezentaci, abychom byly schopni vynásobit jakékoliv matice, při použití tohoto formátu bychom nebyli schopni velkou část matic efektivně reprezentovat, takže tento formát je pro nás taktéž nevhodný. Pro určité konkrétní aplikace, které by však potřebovaly zpracovávat takto specifické matice, by byl tento formát dobrou volbou.

7	0	0	*5*
0	*1*	0	0
8	0	*2*	0
0	*3*	0	*4*

data

*	*7*	5
*	1	*
8	2	*
3	4	*

offset

-2	0	3

Tabulka 2: Struktura formátu DIA

3.2.3 Formát ELL

Formát ELLPACK/ITPACK [14] ukládá nenulové prvky matice do husté matice *m x k*, kde *k* je délka nejdelšího nenulového řádku v matici. Všechny řádky, které jsou kratší než tento řádek jsou pak ve výsledné matici vyplněny zástupnými symboly (*). Dále je potřeba ke každému nenulovému prvku uložit index sloupce, ve kterém se nachází. V tabulce 3 je vlevo opět uvedena hustá matice, uprostřed se nachází matice s řídkými daty a úplně vpravo matice s indexy sloupců pro jednotlivé položky v matici *data*.

Formát je tedy vhodný pro takové matice, které se skládají z řádků s podobnou hustotou. V opačném případě není paměť efektivně využívána. Jelikož se zde snažíme vytvořit pokud možno co nejobecnější metodu pro násobení, tak musíme i tento formát zamítnout.

7	0	0	*5*
0	*1*	0	0
8	0	*2*	0
0	*3*	0	*4*

data

7	5	*
1	*	*
8	2	*
3	4	*

index sloupce

0	3	*
1	*	*
0	2	*
1	2	*

Tabulka 3: Struktura formátu ELL

3.2.4 Formát CRS

Jako poslední uvádíme formát CRS, který jsme se nakonec rozhodli použít. CRS reprezentuje řídkou matici jedním polem obsahujícím všechny nenulové prvky řídké matice seřazené dle řádku, ve kterém se nacházejí a jedním polem, kde je pro každý tento prvek uložen index sloupce. Dále je potřeba uložit informaci o řádku, ve kterém se daný prvek nachází. Toto je v tomto formátu velmi elegantně řešeno dvěma poli o velikosti rovné počtu řádku, kde jsou v jednom poli uloženy indexy začátků a v druhém poli indexy konců všech řádků do pole s daty. Pro matici o velikost *m x n* je paměťová složitost *2k + 2m* kde *k* je počet nenulových prvků matice (*2m* potřebujeme pro dvě pole indexů začátků a konců řádků, *2k* pro uchování indexů sloupců a skutečných hodnot všech nenulových prvků). Tento formát je tedy paměťově velmi efektivní.

Když se nad touto specifikací zamyslíme, snadno zjistíme, že pole s indexy konců jednotlivých řádků pro některé aplikace nepotřebujeme. V případě, že budeme znát celkový počet nenulových prvků, stačí nám znát pouze indexy začátků jednotlivých řádků. Index konce *n*-tého řádku si můžeme snadno dopočítat z indexu začátku řádku prvku *n+1*. Prázdné řádky pak můžeme reprezentovat hodnotou *-1* v tomto poli.

7	0	0	*5*
0	*1*	0	0
8	0	2	0
0	*3*	0	*4*

data	7 5 1 8 2 3 4
index sloupce	0 3 2 0 2 1 3
indexy počátků řádků	0 2 3 5
indexy konců řádků	1 2 4 6

Tabulka 4: Struktura formátu CRS

Tento formát vybíráme zejména pro jeho univerzálnost a pro nízkou pamětovou náročnost. Dále má tento formát jednu velmi zajímavou vlastnost - v případě, že pole indexů sloupců zaměníme za pole s indexy řádků a naopak, získáme transponovanou matici. Jsme tedy schopni vytvořit transponovanou matici bez jakýchkoliv operací. Tohoto lze velmi dobře využít například při výpočtu matice podobnosti. Takto transponovaný formát označujeme jako CCS.

3.2.5 Algoritmus násobení

Ideálně bychom samozřejmě chtěli, aby celý algoritmus proběhl najednou ve stylu černé skříňky. Budeme tedy chtít, aby algoritmus proběhl následujícím způsobem:

1. načtení veškerých potřebných dat,
2. nahrání veškerých dat na grafickou kartu,
3. spuštění kernelu s výpočtem,
4. zkopírování výstupních dat do hostujícího procesu.

Takovýto průběh programu nebudeme schopni z různých důvodů dodržet vždy. Často budeme velmi omezováni velikostí paměti grafické karty, neschopnosti běhového prostředí vytvářet dynamické datové struktury a zejména neschopností provádět klasické universální kritické sekce.

V tomto případě se nám však tímto ideálním způsobem algoritmus zrealizovat podařilo.

Popis algoritmu:

- *Vstupy:*
  - Řídká matice A reprezentována pomocí CRS
  - Hustý vektor x
- *Výstupy:*
  - Hustý vektor y, definovaný jako $y = Ax+y$

Už ze samotného popisu algoritmu je vidět, že nebude pro grafickou kartu nikterak paměťově náročný. Řídkou matici jsme paměťově schopni efektivně reprezentovat pomocí CRS a kromě ní budeme potřebovat již jen dva husté vektory. Celkově bude paměťová složitost pouze $2k + 2m + 2n$, kde k je počet nenulových prvků matice A a m x n je velikost husté matice A.

Algoritmus 3 Násobení řídké matice hustým vektorem

```
function matrix_csr_kernel(
  Řídká matice A,
  Hustý vektor x,
  Hustý vektor y)
begin
  // Zjistí index přiřazený aktuálnímu vláknu
  IndexŘádku = ZjistiAktualníIndex()
  if(IndexŘádku je platný)
    // Počítáme y = Ax + y. Proto nezačneme počítat sumu od nuly, ale od prvku y
    Suma = y[IndexŘádku]

    // Zjistíme indexy konce a začátku aktuálního řádku v matici A
    IndexZačátku = PočátekŘádku(A, IndexŘádku)
    IndexKonce = KonecŘádku(A, IndexŘádku)

    // Provedeme samotné násobení
    for (i = IndexZačátku; i <= IndexKonce; i++)
      // Pro každé i se provede vynásobení jednoho prvku matice A
      // a přičtení výsledku do y.

      // První musíme zjistit index sloupce do kterého aktuální prvek náleží.
      IndexSloupce = ZjistiIndexSloupce(A, i);

      // Teď můžeme provést vynásobení a přičtení.
      Suma += Hodnota(A, i) * x[IndexSloupce]
    endfor
  endif
  // Zapíšeme do y výsledek.
  y[IndexŘádku] = Suma
end
```

Na algoritmu 3 vidíme algoritmus výpočtu pro jedno vlákno běžící na grafické kartě. Toto vlákno poté spustíme pro každý řádek matice A, pro který algoritmus provede vynásobení a přičtení do vektoru y. Jak vidíme, algoritmus je díky vybrané reprezentaci řídké matice poměrně jednoduchý.

3.3 Násobení řídké matice řídkou maticí

Při řešení této části jsme využili velkou spoustu poznatků získaných z násobení řídké matice hustým vektorem. Pro reprezentaci matice jsme opět vybrali formát CRS, který se osvědčil.

3.3.1 Algoritmus násobení

Během návrhu algoritmu jsme přemýšleli nad několika možnými implementacemi. Jednou z možností implementace je rozdělit problém násobení matic na násobení řádkových a sloupcových vektorů daných matic. Bohužel jsme však při implementaci narazili hned na několik problémů vycházejících přímo z architektury nVidia CUDA, které nám znemožnili vytvořit efektivní kernel. Značným problémem je zejména velký počet prázdných smyček, které by program musel vykonat a nutnost častého volání kernelu.

První návrh nás ale přivedl na jiné řešení, veškerá potřebná data budou nakopírována přímo na grafickou kartu, kde každé jednotlivé spuštěné vlákno spočítá výslednou hodnotu jednoho prvku matice. Celkový průběh algoritmu jsme tedy navrhli takto:

- na grafickou kartu budou nakopírována veškerá potřebná data,
- kernel bude spuštěn jednou pro každý jednotlivý prvek *husté* matice,
- kernel ze vstupních dat spočte hodnotu jemu přiřazenému prvku husté matice,
- z grafické karty bude vykopírován výsledek.

Tímto postupem zároveň docílíme toho, že celá matice bude spočtena v jednom běhu kernelu.

Popis algoritmu:

- *Vstupy:*
  - Řídká matice A reprezentována pomocí CRS
  - Řídká matice B reprezentována pomocí CCS
- *Výstupy:*
  - Hustá matice C definovaná jako $C = A \times B$

A kompletní algoritmus vykonávaný kernelem uvádíme na algoritmu 4.

3.3.2 Spojení s Cannonovým algoritmem v jednoprocesorovém prostředí

Výše uvedený návrh algoritmu pro násobení dvou řídkých matic má mnoho výkonnostních výhod, ale také jeden nepříjemný problém. Nepodařilo se nám totiž žádným způsobem obejít neschopnost CUDA pracovat s dynamickými datovými strukturami a museli

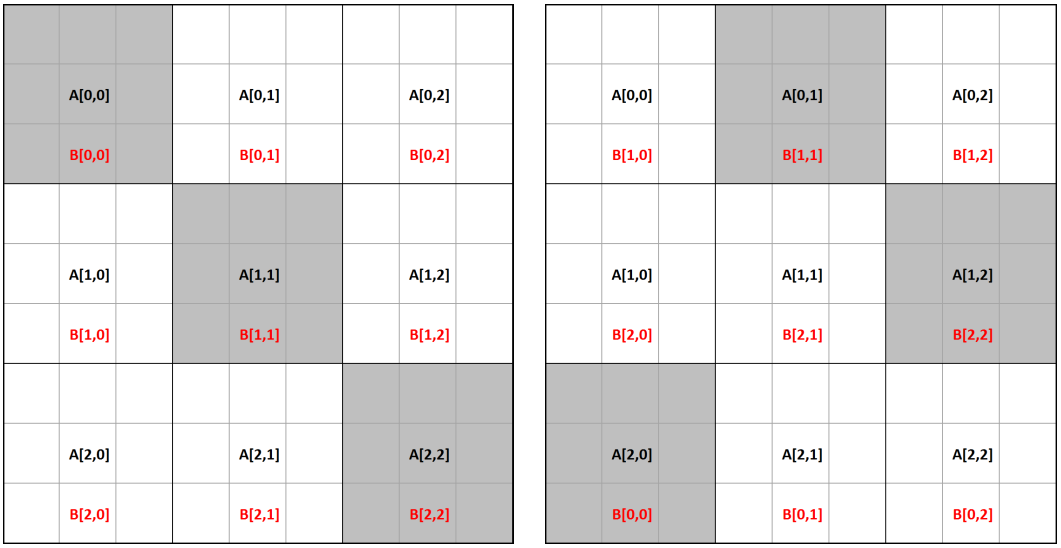
Algoritmus 4 Násobení řídké matice řídkou maticí

```
function matrix_csr_ccs_kernel(
  Řídká matice A ve formátu CRS,
  Řídká matice B ve formátu CCS,
  Hustá výstupní matice C)
begin
  // Zjistí indexy do dat přiřazené aktuálnímu vláknu
  IndexŘádku = IndexZpracovávanéhoŘádku()
  IndexSloupce = IndexZpracovávanéhoSloupce()
  if(IndexŘádku >= počet řádků husté matice C) return
  if(IndexSloupce >= počet sloupců husté matice C) return
  if(DélkaŘádku(IndexŘádku) == 0 && DélkaSloupce(IndexSloupce) == 0) return

  Suma = 0

  // Proměnnou i použijeme jako index do hodnot matice A, j pro indexy matice B
  for(
    i = IndexPočátkuŘádku(A, IndexŘádku),
    j = IndexPočátkuSloupce(B, IndexSloupce);
    j <= IndexKonceSloupce(B, IndexSloupce) && i <= IndexKonceŘádku(IndexŘádku);
    )
    // Posuneme se s indexem sloupce
    if(IndexSloupce(A, i) < IndexŘádku(B,j))
      i++
    // Posuneme se s indexem řádku
    else if(IndexSloupce(A, i) > IndexŘádku(B,j))
      j++
    // Indexy se shodují, provedeme násobení a posuneme se
    else
      sum += Hodnota(A, i)* Hodnota(B, j);
      i++
      j++
    endif
  endfor

  // Nastavíme výslednou hodnotu matice C
  Hodnota(C, IndexŘádku * počet sloupců husté matice C + IndexSloupce) = Suma
end
```



První krok algoritmu *Druhý krok algoritmu*

Obrázek 3: Cannonův algoritmus - příklad rozdělení matice

jsme pro reprezentaci a přenesení výsledku násobení použít hustou matici. Toto je limitujícím faktorem zejména pro matice s nízkou hustotou, které jsou samy o sobě na paměťový prostor nenáročné, ale v momentě, kdy použijeme tento algoritmus a výsledek na grafické kartě uchováváme v hustém formátu, tak paměťové nároky rychle vzrostou a brzy se dostaneme na velikost matice, kterou již nebudeme schopni uchovat v paměti grafického akcelerátoru a nebudeme ji schopni vynásobit.

Řešení, které zde prezentujeme, je spojení tohoto algoritmu s Cannonovým algoritmem. Ten je sice navržen a nejlépe funguje v distribuovaném prostředí, ale pro svoje nízké paměťové nároky je pro náš algoritmus velmi vhodný. V následujícím odstavci je popsán princip tohoto algoritmu.

Cannonův algoritmus pracuje velmi jednoduchým způsobem. Matici A i matici B rozdělí na čtvercové bloky. Naše matice o velikosti $m\ x\ n$ bude rozdělená na $p\ x\ p$ bloků. Algoritmus je původně navržen pro distribuovaná prostředí a počítá s tím, že každý takto vytvořený blok bude zpracováván jedním spuštěným procesem. Příklad rozdělení matic o velikosti $9\ x\ 9$ vidíme na obrázku 3. Vyznačené prvky na diagonále mají speciální význam, jedná se o prvky, které algoritmus považuje za počáteční. V prvním kroku procesy zpracovávající tyto prvky rozešlou své části matice A svým sousedům na řádku, každý proces tedy obdrží matici A_2. Všechny procesy nyní provedou vynásobení A_2 se svým lokálním blokem matice B a výsledek přičtou do lokálně uložené části matice C. V následujícím kroku zrotujeme celou matici B o jeden blok směrem nahoru a posuneme bloky matice A, které považujeme za startovní o jeden blok doprava. Celý proces opakujeme dokud nedostaneme matici B zpět do své původní podoby. Po skončení algoritmu získáme na každém procesu v lokálně uloženém bloku C část výsledku násobení. Po spo-

jení těchto částí dostaneme kompletní výsledek. Příklad prvních dvou kroků je rozkreslen na obrázku 3. Celý algoritmus uvádíme rozepsaný v algoritmu 5.

Algoritmus 5 Cannonův algoritmus - distribuované prostředí

```
function Cannon(
   Matice A,
   Matice B,
   Matice C)
begin
   Proces = TentoProces()
   if Proces je master
      // Vytvoření procesu pro každý blok
      VytvořProcesy()
      // Počáteční rozdělení matic A, B a C na bloky
      RozdělMatici()
      // Rozešli bloky odpovídajícím procesům
      RozešliBloky()
   endif

   // Příjem bloků pro tento proces
   BlokA = PřijmyA()
   BlokB = PřijmyB()
   BlokC = PřijmyC()

   p = Počet bloků v jedné úrovni
   IndexŘádku = IndexŘádkuProcesu(Proces)
   IndexSloupce = IndexSloupceProcesu(Proces)
   for(i = 0; i < ; i++)
      if (IndexŘádku + i) % p == IndexSloupce
         // Proces bude vysílat
         RozešliProcesůmNaŘádku(BlokA)
         BlokC += BlokA * BlokB
      else
         // Proces bude příjmat
         TmpA = PřijmyBlokAOdAktivníhoŘádku()
         BlokC += TmpA * BlokB
         ZrotujMaticiNahoru(B)
      endif
   endfor
end
```

Vidíme tedy, že k vynásobení jednoho bloku matice potřebujeme uchovávat pouze jeden blok z každé matice, kterou násobíme, jeden blok matice, kterou jsme obdrželi od souseda a jeden blok matice, do které budeme zapisovat výsledek. Zároveň vidíme, že

Algoritmus 6 Cannonův algoritmus - sekvenční implementace

```
function cannon(
  Řídká matice A ve formátu CRS,
  Řídká matice B ve formátu CCS,
  Hustá výstupní matice C)
begin
  BlokyA[] = RozdělMatici(A)
  BlokyB[] = RozdělMatici(B)
  p = Počet bloků v jedné úrovni
  for(int i = 0; i < p; i++)
    for(IndexŘádku = 0; IndexŘádku < p; IndexŘádku++)
      // Index pro aktualní řádek + číslo řádku (pro prvek na diagonále) + i pro rotace a
      IndexA = (IndexŘádku * p + IndexŘádku + i) % p
      BlokA = BlokyA[IndexA]
      IndexŘádkuB = ((IndexŘádku + i) * p) % p
      for(IndexSloupce = 0; IndexSloupce < p; IndexSloupce++)
        // Dojde k násobení vybraného A a B
        IndexB = IndexŘádkuB + IndexSloupce
        BlokB = BlokyB[IndexB]
        matrix_csr_ccs_kernel(BlokA, BlokB, BlokC)
      endfor
    endfor
  endfor
end
```

při násobení dochází k velké recyklaci bloků, což nám pomůže ušetřit velkou část komunikace s grafickou kartou.

Nejdříve popíšeme jak by mohl tento algoritmus vypadat na straně hostujícího procesu, který bude spravovat jednotlivé bloky matic a zasílat požadavky na zpracování jednotlivých bloků na grafickou kartu.

Samozřejmě bychom mohli naimplementovat algoritmus v MPI nebo v PVM a spouštět procesy na jedné stanici a nemuseli bychom vymýšlet sekvenční implementaci, ale tato implementace by byla velmi neefektivní. Využívána bude jenom jedna grafická karta, procesy by musely čekat, až tato bude uvolněna a implementace několika procesy by byla zbytečně degradována. Proto jsme vytvořili sekvenční jednoprocesovou variantu, její podrobný popis vidíme na algoritmu 6.

Na straně grafické karty můžeme s tímto návrhem využít úplně stejného kernelu, jaký jsme navrhli v sekci 3.3.1.

Touto implementací odstraňujeme problém omezené paměti grafické karty a jsme schopni akcelerovat násobení libovolně velké řídké matice. Efektivita zvoleného postupu bude ověřena v sekci 4.

3.3.3 Využití v distribuovaném prostředí

Zároveň tím, že celá implementace je postavena nad Cannonovým algoritmem, umožňujeme snadné přenesení celého algoritmu do distribuovaného prostředí. Případné začlenění našeho kernelu do distribuované implementace např. v MPI nebo PVM by byla velmi jednoduchá a umožnila by, při existenci clusteru vybaveného grafickými kartami, dosáhnout mnohonásobného zrychlení. O MPI se lze dočíst více na domovské stránce [15], o PVM pak např. v knize [16].

Implementace je tedy do budoucna zajímavá a spolu s hardwarem velmi dobře škálovatelná.

3.4 Výpočet matice podobnosti pro matice s velmi nízkou hustotou

Jedním z problémů, na které jsme mysleli, při navrhování těchto algoritmů, je výpočet matice podobnosti. Pro specifické případy, kdy by matice byla velmi řídká, by zde prezentované implementace začaly ztrácet na efektivitě. Na vině je zejména nutnost uchovávat výsledky výpočtů v husté matici, která tak zbytečně zabírá vekou část paměti na grafické kartě.

Výpočet matice podobnosti je definován jako násobení matice A transponovanou maticí A^T. Pro toto jsme tedy naimplementovali alternativní algoritmus. Jeho implementace je založena na paralelizaci sekvenčního algoritmu, který postupuje v matici A řádek po řádku, pro každý prvek tohoto řádku vyhledá v matici A^T řádek se stejným indexem, jako je index sloupce tohoto prvku z matice A. Hodnotou z matice A pak pronásobíme hodnoty z řádku matice A^T a výsledky postupně přičítáme do hustého vektoru dle indexu sloupce prvku z matice A^T. Výsledkem zpracování jednoho řádku je vždy kompletní řádek výsledné matice. Celý algoritmus je patrnější ze zápisu v algoritmu 7. Sekvenční algoritmus jsme převzali z řešení prezentovaných v publikacích [17] a [18].

Naší ideou je rozdělit algoritmus tak, aby v běhu jednoho kernelu mohl zpracovat několik takovýchto řádků najednou. V případě, že bychom naivně paralelizovali algoritmus pouze na úrovni jednoho řádku, tak bychom narazili na velkou režii datové komunikace, která by byla nutná při začátku zpracování každého řádku.

V paměti grafické karty bychom řídkou matici A^T udržovali neustále a měnili jenom potřebné řádky matice A. Kernel bychom potom spustili jednou pro každý prvek všech nahraných řádků matice A a nechali jej spočítat výsledek pro tento konkrétní prvek. Když se podíváme na algoritmus 7 podrobněji, tak zjistíme, že by mohlo dojít, při námi navrhované paralelizaci, k souběžnému pokusu o přičtení hodnoty do výstupního vektoru. Toto naštěstí lze ošetřit atomickou operací přičtení, která je na CUDA realizovatelná.

Vzhledem k faktu, že počítáme s velmi nízkou hustotou matice A^T, budeme schopni udržet v paměti grafické karty i velmi velké matice. Pro případ, kdy by se matice A^T i tak nebyla schopna vejít do této paměti, museli bychom do budoucna zvažovat kombinaci s některým z algoritmů, které dokáží násobení matice rozdělit na části, podobně jako jsme to udělali v sekci 3.3.2 s násobením dvou řídkých matic a s Cannonovým algoritmem.

Algoritmus 7 Výpočet matice podobnosti pro matice s velmi nízkou hustotou

```
function similarity_matrix(
   Řídká matice A ve formátu CRS,
   Řídká matice B (transponovaná A) ve formátu CRS,
   Výstupní hustá matice C
)
begin
   foreach(Řádek r matice A)
      // Hustý vektor pro výsledek jednoho řádku o velikosti rovné husté šířce matice B
      Výsledek[Hustá šířka matice B]
      foreach(Prvek p v Řádku r)
         TransponovanýŘádek = B[IndexSloupce]
         foreach(Prvek tp v TransponovanémŘádku)
            Výsledek[IndexSloupce(tp)] += Hodnota(p) + Hodnota(tp)
         endforeach
      endforeach
      VložŘádekDoMatice(C, Výsledek)
   endforeach
end
```

Algoritmus 8 Výpočet matice podobnosti pro matice s velmi nízkou hustotou - kernel

```
function similarity_matrix_kernel(
   Řídká matice A ve formátu COO,
   Řídká matice B (transponovaná A) ve formátu CCS,
   Hustá matice C pro výstup
)
begin
   Index = AktuálněZpracovávanýIndex()
   IndexSloupceA = IndexSloupce(A, Index)
   IndexŘádkuA = IndexŘádku(A, Index)
   ZačátekŘádkuB = ZačátekŘádku(B, IndexSloupceA)
   KonecŘádkuB = KonecŘádku(B, IndexSloupceA)
   DélkaŘádkuB = KonecŘádkuB - ZačátekŘádkuB + 1

   if(DélkaŘádkuB > 0)
      for(i = ZačátekŘádkuB; i < KonecŘádkuB; i++)
         IndexSloupceB = IndexSloupce(B, i)
         Hodnota(C, IndexŘádkuA * PočetŘádkůA + IndexSloupceB) +=
                  Hodnota(A, Index) * Hodnota(B, i)
      endfor
   endif
end
```

Popis kernelu pro CUDA:

- *Vstupy:*
  - d řádků matice A reprezentovaných pomocí formátu COO
  - Řídká matice B (transponovaná A) reprezentována pomocí CCS
- *Výstupy:*
  - Část husté matice C definované jako $C = A \; x \; B$

Kód hostujícího procesu vykonává pouze smyčku přes jednotlivé řádky matice A a stará se o zajištění vykopírování d řádků ke zpracování do formátu COO. Kompletní kód je samozřejmě přiložen s testovacím programem. Pseudokód kernelu uvádíme v algoritmu 8.

Naše implementace umožňuje nastavit počet najednou zpracovávaných řádků matice. Toto je parametr, který bude mít velký vliv na jeho výkonnost a před řešením konkrétní úlohy bude třeba zvážit jeho nastavení.

4 Experimenty s řídkými maticemi

V této kapitole popisujeme dva experimenty, které jsme provedli s námi vytvořenými algoritmy pro zpracování řídkých matic. Naším cílem je zejména otestovat rychlost těchto algoritmů vůči jejich referenčním sekvenčním implementacím.

Všechna měření jsme provedli na této konfiguraci:

- AMD Phenom II X4 940
- 8GB PC1066 DDR2 RAM
- nVidia GeForce GT250 (128 stream procesorů)

4.1 Násobení řídké matice řídkou maticí

Pro ověření algoritmů popsaných v sekci 3.3 jsme provedli sérii experimentů nad daty vygenerovanými pomocí nástrojů blíže popsaných v sekci 7.5.

Nejprve jsme naměřili výsledky implementace uvedené v podkapitole 3.3.1 a postavili jsme jej proti sekvenční verzi násobení řídkých matic, která principiálně postupovala podobným způsobem jako naše implementace v CUDA. Její kompletní implementace je samozřejmě dodána v testovacím programu.

Námi prezentované spojení s Cannonovým algoritmem sepsané v sekci 3.3.2 pak umožňuje velkou variabilitu v nastavení velikosti zpracovávaných bloků, které ovlivní jak náročnost zpracování jednoho bloku, tak počet potřebných násobení k obdržení výsledků, proto jsme vždy experimentálně ověřili vývoj výkonnosti tohoto algoritmu v závislosti na těchto nastaveních.

Nejprve jsme provedli sérii měření s maticemi několika velikostí s 50% hustotou. Výsledek těchto měření shrnujeme v tabulce 5. Ve sloupci *Referenční čas [ms]* uvádíme naměřený čas sekvenční implementace, ve sloupci *CUDA [ms]* je uveden čas implementace ze sekce 3.3 a v posledním sloupci *Zrychlení [%]* je uvedeno procentuální zrychlení definováno jako $Zrychlení = \frac{Referenční\,čas}{Implementace\,v\,CUDA} \cdot 100$.

Velikost husté matice	Hustota	Referenční čas [ms]	CUDA [ms]	Zrychlení [%]
1000x1000	50%	4 923	780	631
2000x2000	50%	40 230	7 180	560
3000x3000	50%	127 421	29 070	438
4000x4000	50%	322 251	77 531	416
5000x5000	50%	698 370	167 217	418
6000x6000	50%	1 068 154	303 656	352

Tabulka 5: Experiment - matice s 50% hustotou

Vidíme, že zrychlení není oproti referenčním časům konstantní, ale poměrné zrychlení se stále snižuje. Toto je dáno zejména nutností udržovat v paměti grafické karty blok dat o velikosti husté matice pro zápis výsledku a jeho nutný přenos. Výkonnost také bude klesat s počtem smyček, které musí jeden kernel vykonat, proto se naše volba spojení s Cannonovým algoritmem a volba cesty většího počtu násobení menších bloků ukazuje být vhodnou.

Dále jsme na těchto maticích proměřili výkonnost implementace uvedené v 3.3.2 (spojení s Cannonovým algoritmem). V tabulce 6 shrnujeme výsledek těchto měření. Ve sloupci *CUDA [ms]* uvádíme naměřené časy z implementace bez použití Cannonova algoritmu a ve sloupci *Cannon 1/4[ms]* uvádíme čas implementace s použitím Cannonova algoritmu a rozdělením matice na 4 čtvercové podbloky. Obdobně je ve sloupci *Cannon 1/16[ms]* uveden čas pro implementaci s Cannonovým algoritmem při rozdělení vstupní matice na 16 čtvercových podbloků, stejné platí pro další sloupce. Nejlepší výsledek Cannonova algoritmu pro danou matici je vyznačen tučně. Ve sloupci označeném jako *Zrychlení* uvádíme dosažené zrychlení Cannonova algoritmu s nejlepším výsledkem oproti předchozím naměřeným časům našeho algoritmu v CUDA. V posledním sloupci *Absolutní zrychlení* uvádíme celkové zrychlení dosažené oproti referenční implementaci. Pro přehlednost je tabulka rozdělena na dvě části.

Vidíme, že se nám tímto spojením podařilo docílit ještě vyššího urychlení výpočtu než prostým vynásobením těchto matic.

Na závěr jsme ještě prověřili vývoj výkonu s měnící se hustotou matice. Pro tento účel jsme vygenerovali matici o velikosti 3000 x 3000 prvků s různými hustotami. Výsledky prezentujeme v tabulce 7. Význam sloupců tabulky je obdobný jako v předchozím případě. Nejlepší dosažená hodnota Cannonova algoritmu je opět vyznačena tučně. Ve sloupci *Zrychlení* uvádíme dosaženého zrychlení jednoduché implementace v CUDA oproti referenčnímu času a ve sloupci *Absolutní zrychlení* uvádíme nejvyšší dosažené zrychlení oproti referenčnímu času. Ať už dosažené Cannonovým algoritmem nebo prostým algoritmem.

Toto měření potvrdilo naše domněnky z předchozích testů. Dále se ukazuje, že implementace s Cannonovým algoritmem zvládá velmi dobře i řešení úloh s maticemi s velmi nízkou hustotou. U těchto matic ale, vzhledem k nutnosti uchovávat výsledek hustě, dochází ke snižování počtu vynásobených prvků v rámci jednoho běhu kernelu. Jsme nuceni násobit zbytečně malé bloky matice, protože větší do paměti grafické karty nenahrajeme. To v důsledku znamená velký počet násobení jednotlivých bloků navíc.

Proto jsme přišli, alespoň pro výpočet matice podobnosti, s alternativním algoritmem popsaným v sekci 3.4.

Velikost matice	Hustota	CUDA [ms]	Cannon 1/4[ms]	Cannon 1/16[ms]	Cannon 1/25[ms]
1000x1000	50%	780	**936**	1 326	1 701
2000x2000	50%	7 180	**4 820**	5 397	4 927
3000x3000	50%	29 070	16 271	13 775	**13 541**
4000x4000	50%	77 531	41 558	32 230	**30 482**
5000x5000	50%	167 217	89 763	68 921	**59 858**
6000x6000	50%	303 656	168 528	107 048	103 541

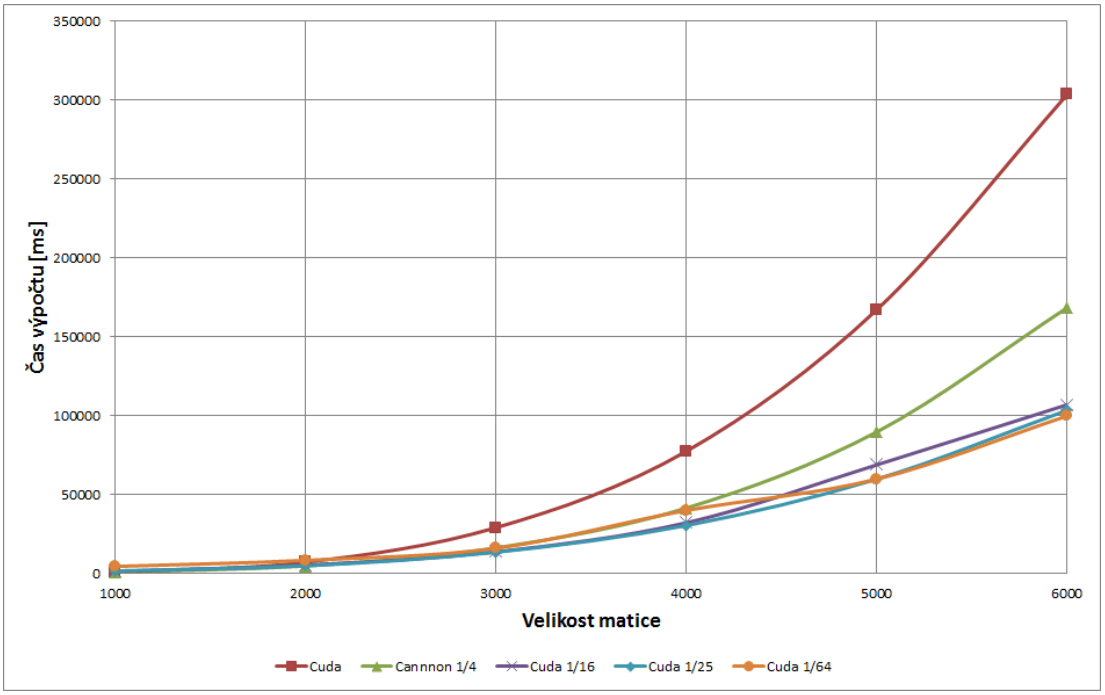
Velikost matice	Hustota	CUDA [ms]	Cannon 1/64[ms]	Zrychlení [%]	Absolutní zrychlení [%]
1000x1000	50%	780	4 555	-120	526
2000x2000	50%	7 180	4 972	149	835
3000x3000	50%	29 070	16 021	215	941
4000x4000	50%	77 531	40 007	254	1 057
5000x5000	50%	167 217	59 873	279	1 167
6000x6000	50%	303 656	**99 903**	304	1 069

Tabulka 6: Experiment - matice s 50% hustotou s Cannonovým algoritmem

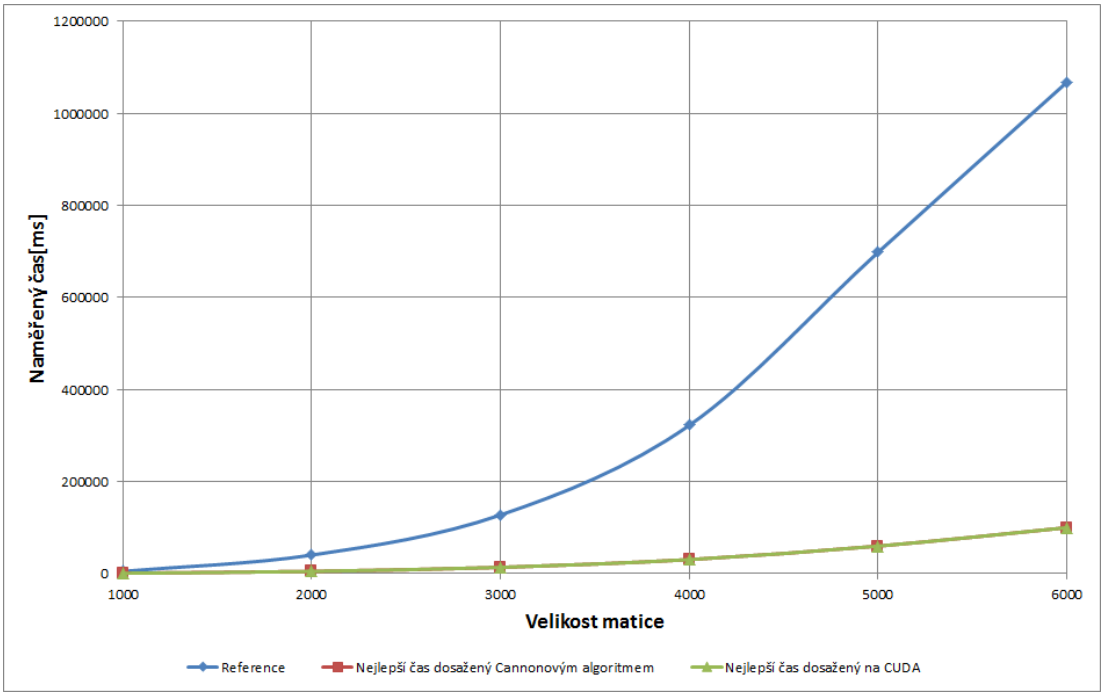
Hustota	Reference [ms]	CUDA [ms]	Cannon 1/4 [ms]	Cannon 1/16[ms]	Cannon 1/25 [ms]
5	13 393	271	**374**	718	1 108
10	27 635	757	**764**	1 030	1 373
15	41 067	1 818	**1 482**	1 732	1 965
20	54 382	3 682	**2 450**	2 574	2 932
25	68 024	5 554	3 963	**3 760**	3 775
30	81 588	9 930	5 881	4 929	**4 821**
35	94 872	12 074	7 551	7 503	**6 802**
40	107 212	21 778	11 591	9 079	**8 393**
45	118 314	27 143	15 023	11 076	**10 218**
50	127 453	29 071	16 271	13 775	**13 541**

Hustota	Reference [ms]	CUDA [ms]	Cannon 1/64 [ms]	Zrychlení [%]	Absolutní zrychlení [%]
5	13 393	271	2 886	4 942	4 942
10	27 635	757	3 291	3 651	3 651
15	41 067	1 818	3 947	2 259	2 771
20	54 382	3 682	4 571	1 477	2 220
25	68 024	5 554	5 616	1 225	1 809
30	81 588	9 930	6 879	822	1 692
35	94 872	12 074	8 954	786	1 395
40	107 212	21 778	9 314	492	1 277
45	118 314	27 143	11 590	436	1 158
50	127 453	29 071	16 021	438	941

Tabulka 7: Experiment - matice s různou hustotou



Obrázek 4: Graf - porovnání jednoduché implementace v CUDA s implementací využívající Cannonův algoritmus



Obrázek 5: Graf - porovnání CUDA implementací s referenčními časy

Naměřené výsledky jsme shrnuli v grafu 5, který ukazuje porovnání časů dosažených na nVidia CUDA s referenčním časem. Série *Reference* reprezentuje referenční čas, druhá série ukazuje nejlepší dosažený čas na CUDA ve spojení s Cannonovým algoritmem a poslední série ukazuje nejlepší možný čas, který jsme dosáhli na CUDA (nejlepší čas běhu algoritmu ze sekce 3.3 nebo 3.3.2). Z tabulky a grafu je patrné, že ve většině případů byl lepší Cannonův algoritmus. Na obrázku 4 pak vidíme v grafu srovnání jednoduché varianty algoritmu s variantou spojenou s Cannonovým algoritmem.

4.2 Výpočet matice podobnosti pro velmi řídká data

V této části budeme prezentovat naměřené výsledky algoritmu sepsaného v sekci 3.4. Tento algoritmus umožňuje pro zvýšení efektivity celého výpočtu nastavit počet zpracovávaných řádků matice v jednom běhu kernelu. Počet takto zpracovávaných řádků budeme dále označovat jako parametr d.

Pro naměření hodnot pro tento experiment jsme použili program popsaný v sekci 7.3, který byl spuštěn na stejné sestavě jako předchozí experiment:

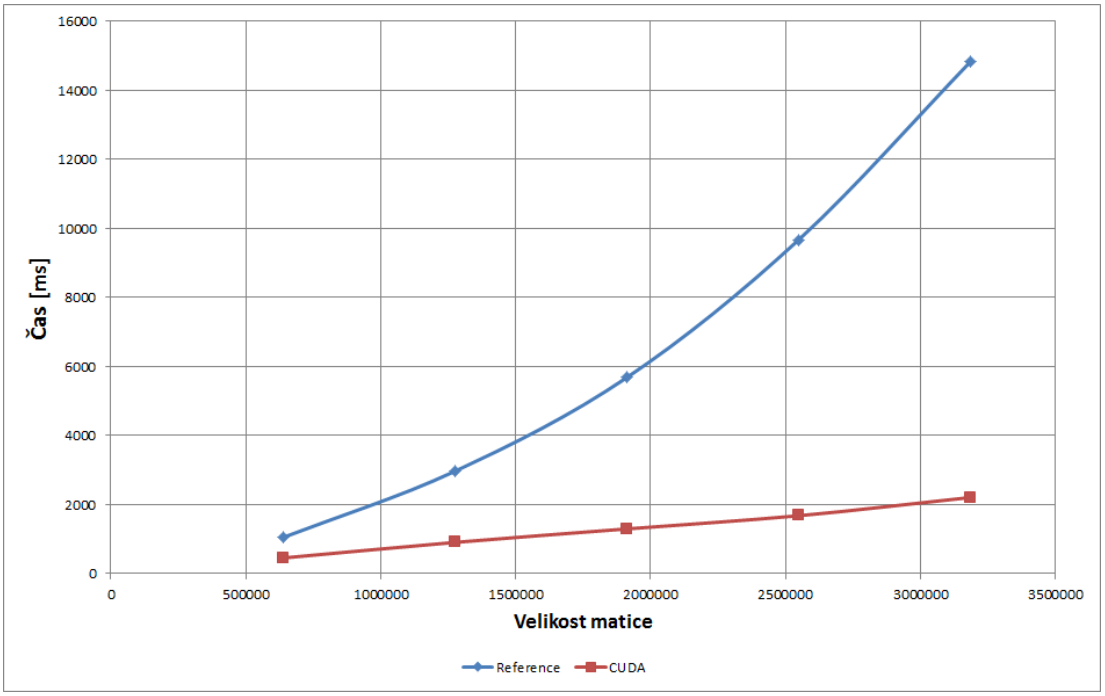
- AMD Phenom II X4 940
- 8GB PC1066 DDR2 RAM
- nVidia GeForce GT250 (128 stream procesorů)

Jako testovací data jsme použili řídkou matici reprezentující kolekci proteinů. Tato kolekce je přiložena na DVD. Pro účely experimentů jsme vytvořili několik kopií této kolekce s různou velikostí, které jsme samozřejmě také přiložili na DVD.

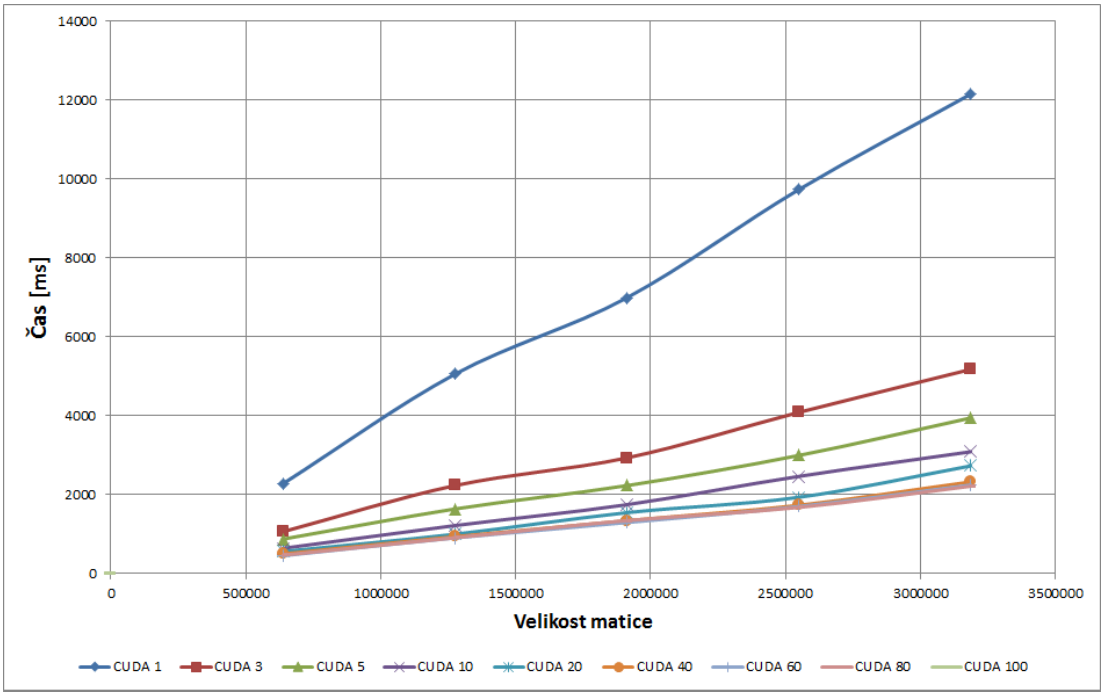
V tabulce 8 shrnujeme naměřené časy. Ve sloupci *Velikost* uvádíme počet nenulových prvků testovací matice. Sloupec *Reference* obsahuje naměřené časy dosáhnuté referenčním algoritmem. Ve sloupci označeném *CUDA 1* uvádíme naměřené časy našeho algoritmu s parametrem $d = 1$, sloupec *CUDA 3* pak obsahuje časy pro $d = 3$, stejné platí pro další sloupce. Nejlepší čas dosažený naším algoritmem je vyznačený tučně. V posledním sloupce *Zrychlení* uvádíme dosažené procentuální zrychlení nejlepšího času algoritmu na CUDA oproti referenční verzi. Tabulka je pro přehlednost opět rozdělena na dvě části.

Naměřené hodnoty jsme dále vykreslili do grafů. Na obrázku 6 jsme zobrazili naměřený referenční čas a nejlepší dosažený čas našeho algoritmu v CUDA. Na druhém grafu na obrázku 7 vidíme potom vývoj jednotlivých časů algoritmu v CUDA v závislosti na nastavení parametru d.

Vidíme, že se zvyšujícím se počtem dat zrychlení roste, algoritmus je tedy obzvláště vhodný pro vyšší objemy dat. Zpočátku jsme měli obavu o rychlost algoritmu, pro jeho realizaci bylo totiž nutné využít operace atomického součtu, která zamkne data, nad kterými pracuje pro ostatní vlákna, provede přičtení a poté data odemkne. Ukázalo se ale, že při přičítání pomocí této operace do pole není zamknuto celé pole, ale pouze jeden konkrétní prvek a v našem konkrétním případě toto nepředstavuje výkonnostní problém.



Obrázek 6: Graf - porovnání referenčního času s algoritmem v CUDA



Obrázek 7: Graf - porovnání časů algoritmu v CUDA v závislosti na parametru d

Velikost	Reference [ms]	CUDA 1 [ms]	CUDA 3 [ms]	CUDA 5 [ms]	CUDA 10 [ms]	CUDA 20 [ms]
637 232	1 045	2 277	1 061	874	640	562
1 274 464	2 964	5 055	2 231	1 638	1 217	999
1 911 696	5 678	6 989	2 933	2 231	1 747	1 545
2 548 928	9 672	9 735	4 087	2 995	2 465	1 934
3 186 160	14 836	12 153	5 177	3 946	3 089	2 730

Velikost	Reference [ms]	CUDA 40 [ms]	CUDA 60 [ms]	CUDA 80 [ms]	CUDA 100 [ms]	Zrychlení [%]
637 232	1 045	499	**453**	468	471	231
1 274 464	2 964	936	**905**	910	917	328
1 911 696	5 678	1 342	**1 295**	1 344	1 373	438
2 548 928	9 672	1 732	1 701	**1 678**	1 764	576
3 186 160	14 836	2 324	2 246	2 216	**2 207**	672

Tabulka 8: Experiment - matice podobnosti pro velmi řídká data

5 Konečné nedeterministické automaty

Stěžejním oborem, který jsme chtěli prozkoumat, je oblast konečných nedeterministických automatů a možnosti jejich implementace na CUDA.

Konečné automaty dnes umožňují efektivně řešit celou řadu výpočetních problému, avšak ve svém základu se nedokáží dobře výkonnostně škálovat na moderním hardware. Důvodem je standardně sekvenční implementace těchto automatů a trend zvyšovat frekvence moderních procesů pomalu a výkon nahrazovat více-jádrovými systémy. Zejména proto by nás zajímala možnost paralelizace automatu.

Implementovali jsme vykonávání obecného nedeterministického konečného automatu na této technologii a funkčnost tohoto jsme otestovali na problému vyhledávání v řetězci s chybou, ke které jsme sestrojili testovací automat.

Celou práci jsme postavili na zkušenostech získaných pří implementaci násobení řídkých matic.

5.1 Definice

Nedeterministický konečný automat (NFA) je dán pěticí *(*Q*,* Σ*,* δ*,* q_0*, F)* kde:

- Q je konečná množina stavů,
- Σ je množina vstupních symbolů,
- δ je přechodová funkce mezi jednotlivými stavy definovaná jako$(\Sigma \bigcup \{\varepsilon\}) \longmapsto P(Q)$,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina konečných stavů.

5.2 Vyhledávání s chybou

Automaty dnes mají široké pole využití, proto nebylo jednoduché vybrat problém, na kterém bychom naši implementaci vyzkoušeli a provedli na ní testy.

Jedním ze zajímavých problémů je vyhledávání v řetězcích s chybou. Toto lze využít při řešení velké spousty úloh, jako je například vyhledávání v sekvencích DNA [19].

Obecně řečeno znamená vyhledávání s chybou nacházení všech výskytů hledaného řetězce x ve vstupním řetězci y. Operace končí vrácením pozic všech částí y, které jsou vzdálené maximálně k kroků od x. Předpokládáme, že $k \in \mathbb{N}$ a $k < |x| \leq |y|$.

Pro měření vzdálenosti k se běžně používají dvě nejznámější vzdálenosti, Levenshteinova vzdálenost a Hammingova vzdálenost. O vyhledávání v řetězcích lze nalézt více informací v [20].

5.2.1 Hammingova vzdálenost

Hammingova vzdálenost mezi dvěma stejně dlouhými řetězci x a y je počet pozic, na kterých se dané dva řetězce neshodují. Můžeme taktéž říct, že jde o počet záměn, které je nutné na jednom řetězci provést, abychom dostali řetězec druhý.

5.2.2 Levenshteinova vzdálenost

Levenshteinova vzdálenost bývá často nazývána *editační vzdálenost* a je definována jako minimální počet operací vložení, nahrazení a smazání jednoho znaku, které jsou potřeba, abychom z řetězce x obdrželi řetězec y. Pro výpočet této vzdálenosti se často postupuje spočtením matice obsahující vzdálenosti mezi všemi jednotlivými prefixy hledaného vzorku a všemi prefixy vstupu. Tato metoda je tedy použitelná pouze v případě, že řetězce jsou poměrně krátké a hodí se tedy spíše k porovnání dvou slov. Konstrukce automatu pro tuto vzdálenost je možná, je popsána např. v [21].

Pro naší další práci jsme si ale vybrali vzdálenost Hammingovu, konstrukce Levehnsteinhova automatu je totiž výrazně komplexnější a byla by nad rámec této práce.

5.3 Konečný automat pro vyhledávání s chybou

Pro tuto kapitolu předpokládejme:

- *y* je řetězec, ve kterém vyhledáváme,
- *x* je vyhledávaný vzorek,
- *k* je maximální vzdálenost hledaného vzorku,
- *m* je délka hledaného vzorku.

Automat vyhledávající v řetězcích s chybou dle Hammingovi vzdálenosti musí tedy rozpoznávat jazyk:

$$A^* \{HammingovaVzdalenost(y, x) \leq k\}$$

Budeme tedy uvažovat automat definovaný takto:

- každý stav automatu je definován jako dvojice *(l, i)* kde *l* je počet aktuálních neshod ve zpracovávaném řetězci a *i* je hloubka stavu, tedy $0 \leq l \leq k$, $-1 \leq i \leq m-1$ a $l \leq i+1$,
- výchozí stav je (0, -1),
- přijímací stavy jsou všechny stavy označené jako (*l*, *m - 1*) pro $0 \leq l \leq k$,
- přechody jsou definovány pro $0 \leq l \leq k$, $0 \leq i < m-1$ a pro $a \in A$ buď ve formě ((0,-1), *a*, (0,-1)) nebo ((*l*,*i*), *x*[*i*+1], (*l*,*i*+1)) nebo ve formě ((*l*,*i*), *a*, (*l*+1,*i*+1)) pokud $a \neq x[i+1]$ a $0 \leq l \leq k-1$.

Konstrukce automatu byla převzata z [20].

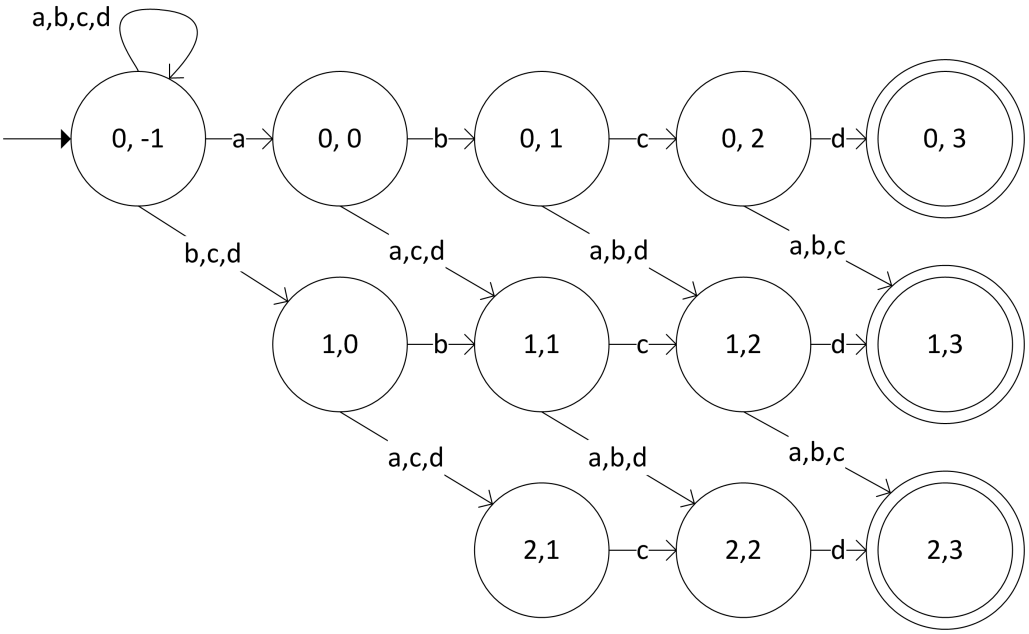
Automat je tedy složen *k*+1 úrovní *l*, kde každá úroveň označuje *l* chyb v aktuálně zpracovaném prefixu hledaného vzorku *x*.

Přechod ((0,-1), *a*, (0,-1)) je jednoduchou smyčkou na prvním stavu automatu zaručující, že automat vyhledá všechny možné výskyty vzorku *x* a ne pouze výskyt první. Tento přechod můžeme rovněž nahradit přechodem ε.

((*l*,*i*), *x*[*i*+1], (*l*,*i*+1)) jsou přechody prováděné v případě shody a přechody ((*l*,*i*), *a*, (*l*+1,*i*+1)) korespondují k nalezeným neshodám.

Automat se skládá z $(k+1) \times \left(m+1-\frac{k}{2}\right)$ stavů a dle [20] jej lze sestrojit v čase $O(k \times m)$.

Jednoduchý automat vyhledávající řetězec *abcd* s maximálně dvěma chybami nad abecedou $A = \{a, b, c, d\}$ můžeme vidět na obrázku 8. U stavu (0,-1) vidíme počáteční smyčku zaručující vyhledání všech výskytů hledaného vzorku. Stavy (0, 3), (1,3) a (2,3) jsou přijímající stavy označující nalezený řetězec na aktuální pozici s žádnou, jednou nebo dvěma chybami. Dále na obrázku vidíme horizontální přechody reprezentující nalezení odpovídajícího znaku a diagonální přechody reprezentující rozdíly mezi řetězci.



Obrázek 8: Automat pro vyhledávání s chybou

5.4 Reprezentace automatu

NFA můžeme reprezentovat několika možnými způsoby. Často bývá pro sekvenční implementace využívána spojová datová struktura, která umožňuje jednoduchou a poměrně efektivní implementaci.

Pro naše účely se však bude lépe hodit reprezentace přechodových stavů pomocí jednoduché binární matice. Binární maticí rozumíme matici obsahující pouze prvky nabývající hodnot 0 a 1.

Binární matice pro náš automat bude vždy čtvercová o velikosti *k x k*, kde *k* je počet stavů automatu. Pro každý symbol ze vstupní abecedy automatu Σ bude existovat jedna taková matice. Buňky reprezentují přechody mezi jednotlivými stavy, přičemž stavy ve sloupcích považujeme za výchozí a stavy na řádcích za cílové.

Příklad jednoduchého automatu reprezentovaného binární maticí vidíme na obrázku 9. Je zde ukázán automat o čtyřech stavech $Q = \{A, B, C, D\}$.

V případě, že se automat nachází ve stavu označeném A a matice na obrázku 9 je maticí odpovídajícímu symbolu na vstupu automatu, pak zjistíme stavy, do kterých automat přejde, vyhledáním všech buněk ve sloupci označeném A majících hodnotu 1. Automat by tedy přešel ze stavu A do stavů A a C.

	A	B	C	D
A	1	0	1	0
B	0	1	0	1
C	1	1	0	0
D	0	0	1	1

Obrázek 9: Automat reprezentovaný binární maticí

5.5 Paralelizace vykonávání automatu

Paralelizace automatu je dlouho řešená otázka. Z existujících řešení jsme nastudovali např. [22], které má tu výhodu, že dokáže paralelně zpracovávat i deterministické automaty. Základní princip tohoto řešení je rozdělení vstupních dat na sebe vzájemně se nepřekrývající bloky. Nad prvním blokem vykonáme standardní běh automatu a zjistíme stav, ve kterém se automat zastaví. Nad každým dalším blokem pak vykonáme paralelně běh automatu tak, že považujeme každý jeho stav za výchozí, obdržíme tak tabulku obsahující konečný stav automatu pro daný blok pro každý stav, ve kterém by automat mohl začít. Pomocí stavu, ve kterém se zastaví první automat, vyhledáme v tabulce druhého automatu koncový stav. Toto opakujeme, než zjistíme koncový stav posledního automatu, který je skutečným výsledkem.

Toto řešení je evidentně stavěno na míru pro systémy posílání zpráv a v nVidia CUDA bychom narazili na dříve popsané problémy. Proto jsme se rozhodli vyzkoušet řešení, které by využilo dříve zjištěných faktů a zkušeností, které jsme získali při implementaci násobení matic.

Naší ideou je vytvořit dva kernely, jeden kernel, který se postará o samotné "vynásobení" binární matice přechodů s vektorem stavů a druhým kernelem, který pak jednoduchým způsobem určí, zda se automat nenachází v některém z koncových stavů a tento fakt pak ohlásí hostujícímu procesu. Takto bychom byli schopni provést implementaci, která by nepotřebovala žádné další synchronizační mechanismy, kromě samotného spuštění kernelu, které bude s hostujícím procesem díky datové komunikaci synchronní.

Kernel pro provedení jednoho kroku automatu by tedy mohl vypadat takto:

Popis algoritmu:

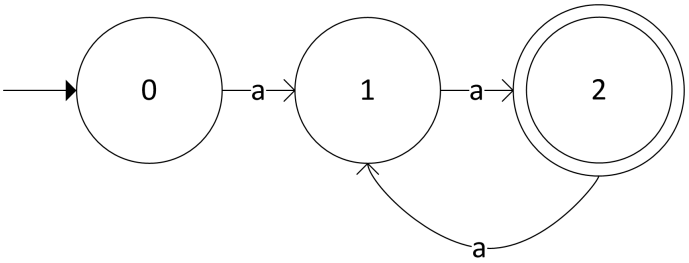
- *Vstupy:*
  - Řídká binární matice A reprezentující přechodovou funkci pro aktuální symbol na vstupu automatu
  - Vektor aktuálních stavů
- *Výstupy:*
  - Vektor aktuálních stavů po provedení přechodu

Pokud bychom ale provedli implementaci kernelu přesně dle tohoto popisu, dostali bychom velké množství datové komunikace mezi hostujícím procesem a grafickou kartou. Proto jsme navrhli řešení, které ideově odpovídá výše uvedenému, ale ulevuje datové komunikaci.

Na grafické kartě budou neustále udržovány dva vektory, které budou uchovávat aktuální stavy a budou pracovat na principu dvojitého bufferu. Jeden vektor bude vždy obsahovat aktuální data a do druhého se bude zapisovat výsledek provedeného přechodu. Po skončení jednoho přechodu dojde k záměně těchto dvou vektorů a vektor určený k zapisování bude znulován.

Dále využijeme druhý kernel, který vektor aktuálních stavů porovná s vektorem konečných stavů, který bude také v paměti akcelerátoru připraven, a hostující proces o případné shodě informuje jednoduchým příznakem.

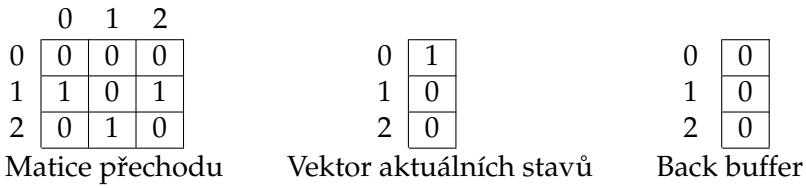
Příklad Pro snažší pochopení našeho řešení uvádíme krátký příklad. Na obrázku 10 vidíme jednoduchý automat rozpoznávající všechna slova obsahující sudý počet symbolů *a*. Předpokládáme množinu vstupních symbolů $\Sigma = \{a\}$.



Obrázek 10: Automat rozpoznávající sudý počet symbolů “a”

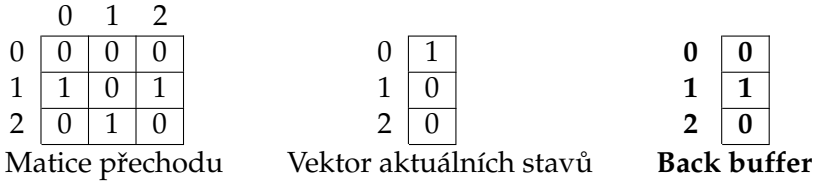
Jediný symbol vstupní abecedy tohoto automatu je právě symbol *a*, matici přechodu pro tento symbol a pro tento automat uvádíme na obrázku 11. Zároveň na obrázku vidíme stav automatu v prvním kroku algoritmu. Hodnota 1 ve vektoru aktuálních stavů reprezentuje aktivní stav, ve kterém se automat nachází. Vidíme, že na začátku se automat nachází ve stavu 0. Dále je na tomto obrázku zobrazen *back buffer*, což je náš vektor

pro zápis stavů, do kterých automat přejde v jednom kroku, v prvním kroku je vynulován.



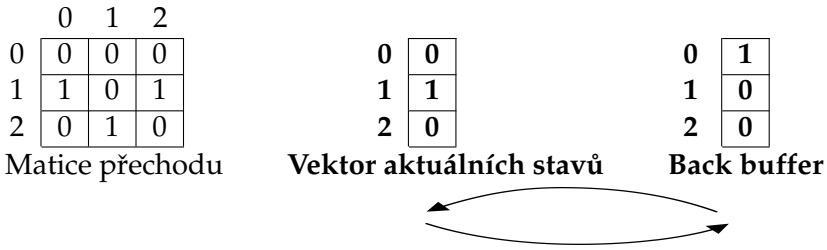
Obrázek 11: Automat rozpoznávající sudý počet symbolů “a” - krok 1

Na obrázku 12 vidíme stav automatu po provedení přechodu pro vstupní symbol *a*, změny jsou vyznačeny tučně. Algoritmus vyhledal v matici přechodu všechny sloupce reprezentující stavy, ve kterých se automat nacházel. V našem případě se jedná pouze o sloupec 0. Z těchto sloupců potom vybral indexy všech řádků, ve kterých je hodnota v matici přechodu rovna jedné. Indexy těchto řádků jsou ve skutečnosti indexy stavů, do kterých má automat přejít. Výsledek si zapíšeme do výstupního vektoru (*back buffer*). V našem případě automat přechází do stavu 1. Tento krok by provedl kernel uvedený na algoritmu 9.



Obrázek 12: Automat rozpoznávající sudý počet symbolů “a” - krok 2

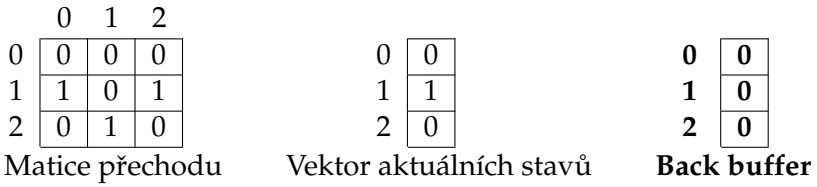
V dalším kroku, zobrazeném na obrázku 13, vyměníme vektor aktuálních stavů za výstupní vektor (*back buffer*), kam algoritmus zapsal výsledek přechodu.



Obrázek 13: Automat rozpoznávající sudý počet symbolů “a” - krok 3

Nyní můžeme porovnat, zda automat nevstoupil do přijímajícího stavu. Toto by provedl kernel uvedený na algoritmu 10. V našem případě víme, že vektor přijímajících stavů je (0,0,1) (můžeme vidět na obrázku 10). Automat tedy do přijímajícího stavu ne-

vstoupil. Nakonec znulujeme náš vektor výstupních stavů, čímž si automat připravíme pro zpracování dalšího symbolu na vstupu. Toto je ilustrováno na obrázku 14.



Obrázek 14: Automat rozpoznávající sudý počet symbolů “a” - krok 4

Celý proces bychom opakovali pro každý další zpracovávaný symbol na vstupu automatu. Řídící algoritmus hostujícího procesu, popsaný tímto příkladem, uvádíme na algoritmu 11.

Chování automatu je možné vždy poupravit pro řešení konkrétních problémů. Např. pro vyhledávání s chybou považujeme počáteční stav automatu vždy za aktivní, tímto docílíme vyhledání všech hledaných vzorků a ne jen jednoho.

Nyní tedy máme návrh, který pro veškerou komunikaci v průběhu procesu potřebuje přenést pouze jeden prvek typu bool a matici přechodů symbolu, který je právě zpracováván. Je třeba si ještě uvědomit, že v případě vyhledávání v delších textech, které je naším cílem, se budou symboly na vstupu mnohokrát opakovat. Tohoto můžeme využít k zachování matic jednotlivých symbolů v paměti grafické karty. Tyto jsou uchovávány řídce a nebudou tedy paměťově náročné. Můžeme je nechat uložené v paměti a pokaždé při spuštění přeposlat kernelu pouze ukazatel do této paměti. Tímto z velké části minimalizujeme potřebnou komunikaci.

Pro úplnost uvádíme popis druhého kernelu pro zjištění, zda se automat nalézá v koncovém stavu:

Popis algoritmu:

- *Vstupy:*
  - Vektor aktuálních stavů x
  - Vektor koncových stavů f
- *Výstupy:*
  - Příznak označující zda automat vstoupil do přijímajícího stavu

Nejdříve uvádíme v algoritmu 9 pseudokód kernelu provádějícího jeden krok automatu.

Algoritmus 9 Automat - kernel pro vykonání jednoho kroku

```
function automata_step(
  Matice přechodu A ve formátu CRS,
  Vektor aktuálních stavů x,
  Vektor stavů po vykonání kroku y)
begin
  IndexŘádku = IndexZpracovávanéhoŘádku()
  IndexSloupce = IndexZpracovávanéhoSloupce()

  if(IndexŘádku < 0 || IndexSloupce < 0) return

  if(x[IndexSloupce] je aktivní)
    y[row_index] = aktivní
  endif
end
```

Dále na algoritmu 10 uvádíme kernel pro zjištění, zda se automat nachází v přijímajícím stavu.

Algoritmus 10 Automat - kernel pro zjištění koncových stavů

```
function automata_check_final_state(
  Vektor aktuálních stavů x,
  Vektor koncových stavů f,
  Výstupní příznak vstupu do přijímacího stavu r)
begin

  Index = AktuálněZpracovávanýIndex()
  IndexPřijímajícíhoStavu = f[Index];
  if(x[IndexPřijímajícíhoStavu] je aktivní) r = true
end
```

Nakonec se podívejme na algoritmus provádějící samotnou logiku spouštění jednotlivých kernelů. V této části se vyskytuje hlavní výkonný kód, který kontroluje dostupnost všech potřebných dat souvisejících s aktuálním krokem automatu na grafické kartě, synchronizuje jednotlivé kroky za pomocí volání kernelu a zpracovává případné nalezené výsledky poskytnuté kernelem. Kompletní výpis vidíme na algoritmu 11.

Algoritmus 11 Automat - hostující proces

```
function automata(
  Automat A,
  Vstupní text Text,
  Výstupní vektor indexů nalezených pozic vzorků o)
begin
  // Příznak který použijeme k identifikaci aktuálního bufferu
  Buffer = 0
  // Naše dva vektory, které budou sloužit jako dva buffery
  define StavyA[] = ZískejPočátečníStavy(A)
  define StavyB[]
  // Příznak, zda jsme vstoupili do přijímajícího stavu
  Ukončit = false
  for(i = 0; i < DélkaTextu(Text); i++)
    // Aktuálně zpracovávaný znak
    c = Text[i]
    // Matice přechodu umístěná v paměti grafické karty
    MaticePřechoduD = ZískejMaticiPřechodu(A, c)
    if(MaticePřechoduD == null)
      MaticePřechoduD = NakopírujDataNaGrafickouKartu(A, c)
    if(Buffer == 0)
      // Jako aktuální vektor stavů použijeme buffer StavyA
      automata_step(MaticePřechoduD, StavyA, StavyB)
      automata_check_final_state(StavyB, ZískejKoncovéStavy(A), Ukončit)
      if(Ukončit)
        Přidej do o index i
      VynulujVektorNaGrafickéKartě(StavyA)
      Buffer = 1
    else if(Buffer == 1)
      // Jako aktuální vektor stavů použijeme buffer StavyB
      automata_step(MaticePřechoduD, StavyB, StavyA)
      automata_check_final_state(StavyA, ZískejKoncovéStavy(A), Ukončit)
      if(Ukončit)
        Přidej do o index i
      VynulujVektorNaGrafickéKartě(StavyB)
      Buffer = 0
    endif
  endfor
end
```

Funkčnost a efektivitu těchto zde prezentovaných algoritmů samozřejmě ověříme experimenty uvedenými v následující kapitole.

6 Experimenty s konečnými nedeterministickými automaty

V průběhu implementace automatu jsme narazili hned na několik problémů způsobených limitacemi technologie nVidia CUDA. Předně jsme byli omezení absencí dynamických datových struktur a standardních kritických sekcí, které způsobili, že jsme byli nuceni navrhnout kernel funkci pouze pro jeden krok automatu a nebyli jsme schopni přinést implementaci, která by byla schopna běh automatu provést v rámci jednoho volání. Toto je samozřejmě velkou limitací výkonu, i tak jsme se snažili přijít s co nejlepší implementací, která by možnosti CUDA využila.

6.1 Metodika měření

Měření probíhalo na stejné sestavě jako měření algoritmů pro násobení řídkých matic. Jedná se o tuto sestavu:

- AMD Phenom II X4 940
- 8GB PC1066 DDR2 RAM
- nVidia GeForce GT250 (128 stream procesorů)

Měření jsme provedli nad jedním souborem reprezentujícím sekvenci DNA o velikosti přibližně 1MB. Nad tímto souborem jsme spustili několik automatů pro vyhledávání s chybou. Naším cílem bylo vyzkoušet více různých kombinací hledaného vzorku a maximálního počtu chyb.

Pro naměření zde prezentovaných hodnot jsme využili program prezentovaný v sekci 7.4. Jako referenční algoritmus jsme použili sekvenční variantu vykonávání automatu založenou na stejném principu jako algoritmus v CUDA.

6.2 Výsledky měření

V tabulce 9 uvádíme naměřené časy referenčního algoritmu a algoritmu napsaného v CUDA. Ve sloupci *Velikost vzorku* uvádíme počet znaků obsažených ve vyhledávaném vzorku. Sloupec *Maximální počet chyb* obsahuje maximální Hammingovu vzdálenost, pro kterou je výskyt vyhledávaného vzorku ještě přijat. Sloupec *Reference* obsahuje naměřené časy referenčního algoritmu a sloupec *CUDA* obsahuje časy algoritmu prezentovaném v sekci 5. Bohužel jsou naměřené časy algoritmu v CUDA výrazně horší než časy referenční.

6.3 Zhodnocení

Vidíme, že bohužel implementace nebyla úspěšná.V rámci rozsahu této práce se nám nepodařilo naimplementovat alternativní algoritmus, který by byl efektivnější. Ve všech případech jsme narazili na různá omezení použité technologie.

Velikost vzorku	Maximální počet chyb	Reference [ms]	CUDA [ms]
10	2	2 231	298 992
10	4	2 465	294 826
10	6	2 496	296 901
20	2	2 433	295 762
20	4	2 574	299 209
20	6	2 621	300 255
20	8	2 654	295 716
20	10	2 593	296 729
50	5	2 730	299 928
50	10	2 855	297 977

Tabulka 9: Experiment - vykonávání konečného deterministického automatu

6.4 Budoucí vývoj

I když implementace nebyla úspěšná můžeme vzhledem k prudkému vývoji technologie CUDA doufat v podporu dynamických datových struktur, jejichž přítomnost by již sama o sobě velmi zefektivnila nejen naši implementaci vykonávání NFA, ale i všechny ostatní zde prezentované algoritmy.

V budoucnu bychom minimálně chtěli vyzkoušet alternativní implementaci, která by vykonávala automat nad několika různými vstupy, která by mohla přispět k výkonnosti.

7 Popis podpůrných programů

V průběhu implementace algoritmů uvedených v této práci vznikly celkem čtyři programy pro otestování jednotlivých algoritmů a dva podpůrné programy určené k generování testovacích dat pro experimenty.

V této sekci jsou podrobně popsány všechny tyto programy. Zároveň jsou s každým testovacím programem dodány vzorová testovací data a *.bat* soubor určený k rychlému spuštění a snadnému vyzkoušení běhu programů.

Všechny testovací programy jsou přiloženy na DVD. Seznam dat a programů přiložených na tomto nosiči je sepsán v příloze B.

7.1 Program pro násobení řídké matice řídkou maticí

Tato testovací aplikace provádí algoritmus popsaný v sekci 3.3. Jedná se o konzolovou aplikaci provádějící násobení matic nahraných ze souborů. Aplikace načte jednu matici ve formátu CRS, transponuje ji (převodem do formátu CCS) a provede samotné násobení.

Po skončení vypíše celkový čas výpočtu. Tato aplikace byla použita pro naměření výsledků experimentu pro násobení dvou řídkých matic ze sekce 4.1.

Seznam všech podporovaných parametrů:

- *m* - řídká matice ve formátu popsaném v příloze A,
- *c* - počet opakování celého výpočtu,
- *r* - pokud je tento parametr uveden, je spuštěna referenční verze bez nVidia CUDA.

Příklad spuštění:

program.exe m a.matrix c 10

Provede deset vynásobení matice a.matrix se svojí transponovanou variantou a vypíše časy jednotlivých násobení na výstup.

7.2 Program pro násobení řídké matice řídkou maticí (Cannon)

Doplňkem pro předchozí aplikaci je tento program, který provede násobení řídkých matic s využitím Cannonova algoritmu. Aplikace načte jednu matici ve formátu CRS, opět ji transponuje, provede násobení a vypíše naměřený čas na výstup.

Seznam všech podporovaných parametrů:

- *m* - řídká matice ve formátu popsaném v příloze A,
- *n* - počet bloků pro rozdělení Cannonovým algoritmem (pro násobení pomocí rozdělení na 4 *x* 4 bloky je *n* rovno 4),
- *s* - velikost jednoho bloku.

Příklad spuštění:

program.exe m 2000_2000_50.matrix n 4 s 500

Provede vynásobení matice 2000_2000_50.matrix (matice o velikosti 2000 x 2000 s 50% hustotou) se svojí transponovanou variantou. Pro Cannonův algoritmus bude tato matice rozdělena na 4 x 4 bloky, každý bude mít velikost 500 x 500. Po skončení násobení zapíše naměřené časy na výstup.

7.3 Program pro výpočet matice podobnosti pro velmi řídké matice

Pro otestování algoritmu prezentovaného v sekci 3.4 jsme vytvořili jednoduchou aplikaci určenou k naměření výsledků tohoto algoritmu.

Seznam všech podporovaných parametrů:

- *m* - řídká matice s proteiny,
- *d* - počet zpracovaných řádků v rámci běhu jednoho kernelu,
- *r* - pokud je tento parametr uveden, je spuštěna referenční verze bez nVidia CUDA.

Příklad spuštění:

program.exe -m indexFileKoh2000.txt d 100

Provede vynásobení matice ze souboru indexFileKoh2000.txt se svojí transponovanou variantou, v každém běhu kernelu zpracuje 100 řádků a naměřený čas běhu vypíše do konzole.

7.4 Program pro vykonávání NFA

Poslední testovací aplikace provádí vykonávání NFA pro vyhledávání v řetězci s chybou nad vstupním textovým souborem. Opět se jedná o konzolovou aplikaci určenou k měření rychlosti vykonávání algoritmu a vypisuje tedy na výstup naměřené časy.

Seznam všech podporovaných parametrů:

- *a* - soubor s automatem popsaný v příloze A,
- *t* - textový soubor se vstupním textem,
- *r* - pokud je tento parametr uveden, je spuštěna referenční verze bez nVidia CUDA.

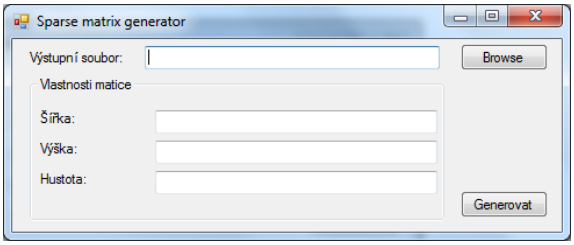
Příklad spuštění:

program.exe -a a.automata -t input.txt

Provede vyhledávání automatem a.automata nad textem v souboru input.txt a vypíše naměřené časy na výstup.

7.5 Program pro generování řídkých matic

Na obrázku 15 vidíme jednoduchou aplikaci, která byla vytvořena k podpoře testování programu pro násobení řídkých matic. Tato aplikace vygeneruje soubor ve formátu popsaném v příloze A. Vyžadováno je zadání výšky a šířky husté matice. Podle dále zadané procentuální hustoty uloží aplikace do výstupního souboru řídkou matici obsahující náhodná čísla.



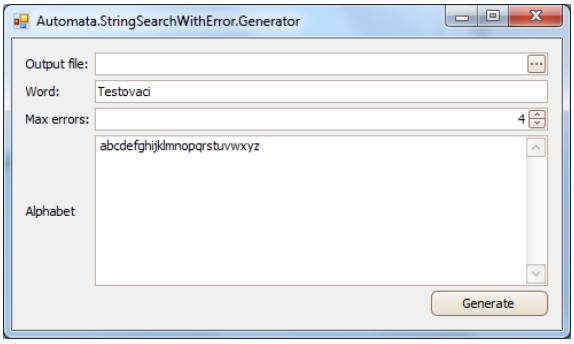
Obrázek 15: Generátor řídkých matic

7.6 Program pro generování NFA

Další aplikací, která byla pro účely testování použita, je generátor nedeterministických konečných automatů vyhledávajících v řetězci s chybou.

- *Ouptut file* - výstupní soubor, do kterého bude automat zapsán
- *Word* - slovo, které bude automat vyhledávat
- *Max errors* - maximální počet chyb, do kterého má automat vyhodnotit výskyt slova jako úspěšný nález
- *Alphabet* - kompletní vstupní abeceda automatu

Aplikace generuje datový soubor podrobněji popsaný v příloze A.



Obrázek 16: Generátor NFA

8 Závěr

V diplomové práci jsme se zabývali implementacemi algoritmů pro násobení řídkých matic a implementací vykonávání obecného konečného nedeterministického automatu na technologii nVidia CUDA. V průběhu tvorby této práce jsme zjistili, že zdaleka ne všechny problémy jsou na této platformě efektivně řešitelné, některé jsou díky limitacím daným touto technologií i zcela neřešitelné.

Seznámili jsme se s prostředím a s vývojem při použití této technologie, dozvěděli jsme se o jejich limitacích a jejich silných stránkách. Při tomto procesu jsme narazili na mnoho chyb, které však jsou v průběhu jejího vývoje opravovány.

V první části zabývající se násobením řídkých matic jsme naimplementovali algoritmus pro násobení řídké matice hustým vektorem, algoritmus pro násobení dvou řídkých matic a algoritmus pro výpočet matice podobnosti pro velmi řídkou matici. Dále jsme program pro násobení dvou řídkých matic zefektivnili spojením s Cannonovým algoritmem, který jej učinil, vzhledem ke své distribuované povaze, do budoucna výkonnostně velmi dobře škálovatelným.

Experimentálně jsme ověřili výkon algoritmu pro násobení dvou řídkých matic, měření prokázalo výrazné urychlení oproti referenční sekvenční implementaci. Experiment propojení tohoto algoritmu s Cannonovým algoritmem dále prokázal další urychlení výpočtu. Řešení, která zde uvádíme jsou velice variabilní a měření prokázaly, že před řešením konkrétní úlohy, je nutné vybrat vhodnou variantu algoritmu a zároveň, pro Cannonův algoritmus, vybrat vhodné nastavení parametrů tohoto algoritmu.

Rovněž jsme docílili urychlení výpočtu matice podobnosti pro velmi řídké matice. Námi prezentovaná implementace je obzvláště vhodná pro větší objemy dat a s narůstajícím počtem nenulových prvků zpracovávané matice zrychlení oproti referenčnímu algoritmu roste.

Vzhledem k omezením plynoucím z podstaty technologie CUDA se nám bohužel nepodařilo vytvořit kvalitní implementaci vykonávání nedeterministického konečného automatu. Podařilo se nám přijít s funkční implementací, která však výkonnostně velmi zaostává.

V rámci práce se objevilo mnoho příležitostí k budoucímu vývoji. Ať už ve zefektivnění algoritmů pro práci s řídkými maticemi nebo pro pokračování v práci na implementaci vykonávání NFA, kde by se v budoucnu spolu s vývojem technologie CUDA mohla situace změnit a i aktuální algoritmus by, při změně několika maličkostí, mohl efektivně fungovat. Mezitím i pro tento problém samozřejmě stále zvažujeme alternativní implementace.

9 Reference

[1] [Online]. Available: http://www.nvidia.co.uk/object/cuda_home_new_uk.html(28.4.2011)

[2] [Online]. Available: http://msdn.microsoft.com/en-us/library/bb219679.aspx(28.4.2011)

[3] [Online]. Available: http://www.opengl.org/documentation/glsl/(28.4.2011)

[4] [Online]. Available: http://msdn.microsoft.com/en-us/library/bb509561.aspx(28.4.2011)

[5] [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf(28.4.2011)

[6] [Online]. Available: http://concurrencykit.org/(28.4.2011)

[7] [Online]. Available: http://www.nvidia.com/object/cuda_gpus.html(28.4.2011)

[8] [Online]. Available: http://developer.nvidia.com/cuda-downloads(28.4.2011)

[9] R. Larson and D. C. Falvo, *Elementary Linear Algebra, Enhanced Edition, 6th Edition*. Cengage Learning, Inc., 2010, no. ISBN-10: 1439044007 ISBN-13: 9781439044001.

[10] R. O. Hill, *Elementary Linear Algebra with Applications, 3rd Edition*. Cengage Learning, Inc., 1996, no. ISBN-10: 0030103479 ISBN-13: 9780030103476.

[11] [Online]. Available: http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUSPARSE_Library.pdf(28.4.2011)

[12] [Online]. Available: http://www.nvidia.com/content/GTC-2010/pdfs/2070_GTC2010.pdf(28.4.2011)

[13] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors."

[14] R.Grimes, D. Kincaid, and D.Young, *ITPACK 2.0 User's Guide*, Center for Numerical Analysis, University of Texas, 8 1979.

[15] [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/index.htm(28.4.2011)

[16] A. Beguelin, J. Dongarra, A. Geist, W. Jiang, R. Manchek, and V. S. Sunderam, *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Network Parallel Computing*. The MIT Press, 1994, no. ISBN-10: 0-262-57108-0 ISBN-13: 978-0-262-57108-1.

[17] P. Dráždilová, J. Dvorský, J. Martinovič, and V. Snášel, "Search in documents based on topical development," *AWIC 2009 Amazon.ca*, no. ISBN-13: 978-3642106866, 2009.

[18] J. Martinovič, V. Snášel, and T. Novosad, "Vector model improvement using suffix trees," *International Journal of Computational Intelligence Research*, vol. 5, pp. 31–40, 2009.

[19] I. Anderson and A. Brass, "Searching dna databases for similarities to dna sequences: when is a match significant?" *BIOINFORMATICS*, vol. 14, pp. 349–356, 1997.

[20] M. Crochemore, C. Hancart, and T. Lecroq, *Algorithms on strings*. Cambridge University Press, 2007, no. ISBN-13: 9780521848992.

[21] K. Schulz and S. Mihov, "Fast string correction with levenshtein-automata," *INTERNATIONAL JOURNAL OF DOCUMENT ANALYSIS AND RECOGNITION*, vol. 5, pp. 67–85, 2002.

[22] J. Holub and S. Štekr, "On parallel implementations of deterministic finite automata."

[23] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*. Morgan Kaufmann Publishers, 2010, no. ISBN: 978-0-12-381472-2.

[24] V. Snášel and T. Koutný, "Algebra of pattern matching automata," Department of Science, Palacký University.

A Popis vytvořených datových formátů

Spolu s generátory byly samozřejmě vytvořeny zvláštní datové formáty, které využívají veškeré programy popsané v této práci. Podrobný popis jejich struktury uvádíme v této kapitole v jednoduchých tabulkách s následující strukturou:

- *pořadí* určuje číslo řádku tabulky a slouží pouze pro orientaci,
- *typ* určuje použitý datový typ pro reprezentaci položky,
- *počet položek* udává kolik konkrétních položek tohoto typu je v souboru za sebou zapsáno,
- *popis* obsahuje vysvětlení významu dané položky.

Je třeba ještě dodat, že všechny zde prezentované formáty jsou jednoduché binární soubory a neměl by být problém je číst ani zapisovat s použitím jakékoliv technologie.

Formát souborů s řídkou maticí

Formát zobrazený v tabulce 10 reprezentuje řídkou matici obsahující prvky typu float uloženou pomocí formátu CRS.

Pořadí	Typ	Počet položek	Popis
1	Int32	1	šířka matice (jedná se o šířku husté matice)
2	Int32	1	výška matice (jedná se o výšku husté matice)
3	Int32	1	počet nenulových prvků matice
4	Float	rovno počtu nenulových prvků matice	data všech nenulových prvků matice
5	Int32	rovno počtu nenulových prvků matice	indexy sloupců pro každý nenulový prvek matice
6	Int32	rovno výšce matice	indexy začátků jednotlivých řádků matice
7	Int32	rovno výšce matice	indexy konců jednotlivých řádků matice

Tabulka 10: Formát souborů s řídkou maticí

Formát souborů s automatem

V tabulce 11 uvádíme formát souboru obecného automatu který byl použit v programech prezentovaných v této práci. Při ukládání matice přechodu pro jednotlivé znaky abecedy byla použita matice ve formátu COO, která byla pro reprezentaci automatu výhodnější než matice ve formátu CRS.

Pořadí	Typ	Počet položek	Popis
1	Int32	1	délka hlavičky souboru
2	Byte	rovno délce hlavičky	ASCII kódovaný řetězec s hlavičkou souboru, prozatím nevyužito, rezervováno pro budoucí případné verzování souboru apod.
3	Int32	1	délka rozpoznávaného výrazu
4	Byte	rovno délce rozpoznávaného výrazu	ASCII kódovaný řetězec rozpoznávaný automatem, prozatím nevyužito, určeno pro orientaci a jednodušší identifikaci činnosti automatu
5	Int32	1	počet přijímajících stavů automatu
6	Int32	rovno počtu přijímajících stavů	indexy všech přijímajících stavů
7	*Následující blok je v souboru zapsán jednou pro každý znak vstupní abecedy automatu*		
7.1	Int32	1	velikost přechodové matice pro znak
7.2	Int32	1	index sloupce nenulového prvku
7.3	Int32	1	index řádku nenulového prvku

Tabulka 11: Formát souborů s automatem

B Obsah přiloženého DVD

Název	Popis
Data	Složka obsahující veškerá data použita pro naměření hodnot uvedených v experimentech
SPMM	Program pro násobení dvou řídkých matic
Source	Zdrojové kódy programu
Bin	Obsahuje přeložený program v Release verzi. Zároveň je přiložen předpřipravený bat soubor, který slouží k ukázkovému spuštění. Pro běh jsou v tomto případě použity soubory ze složky Data
SPMM_Cannon	Program pro násobení dvou řídkých matic s využitím Cannonova algoritmu
Source	Zdrojové kódy programu
Bin	Obsahuje přeložený program v Release verzi. Zároveň je přiložen předpřipravený bat soubor, který slouží k ukázkovému spuštění. Pro běh jsou v tomto případě použity soubory ze složky Data
SPMM_Similarity	Program pro výpočet matice podobnosti pro velmi řídké matice
Source	Zdrojové kódy programu
Bin	Obsahuje přeložený program v Release verzi. Zároveň je přiložen předpřipravený bat soubor, který slouží k ukázkovému spuštění. Pro běh jsou v tomto případě použity soubory ze složky Data
Automata	Program pro vykonávání NFA
Source	Zdrojové kódy programu
Bin	Obsahuje přeložený program v Release verzi. Zároveň je přiložen předpřipravený bat soubor, který slouží k ukázkovému spuštění. Pro běh jsou v tomto případě použity soubory ze složky Data
Matrix Generator	Generátor souborů s řídkou maticí ve formátu CRS
Automata Generator	Generátor souborů s automaty pro vyhledávání s chybou
Diplomová Práce.pdf	PDF soubor s textem této práce